Adobe

Adobe Flex 2

Getting Started with Flex 2

© 2006 Adobe Systems Incorporated. All rights reserved.

Getting Started with Flex 2

If this guide is distributed with software that includes an end-user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end-user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Flex, Flex Builder and Flash Player are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. Windows is either a registered trademark or trademark of Microsoft Corporation in the United States and other countries. All other trademarks are the property of their respective owners.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/). Macromedia Flash 8 video is powered by On2 TrueMotion video technology. © 1992-2005 On2 Technologies, Inc. All Rights Reserved. http://www.on2.com. This product includes software developed by the OpenSymphony Group (http://www.opensymphony.com/). Portions licensed from Nellymoser (www.nellymoser.com). Portions utilize Microsoft Windows Media Technologies. Copyright (c) 1999-2002 Microsoft Corporation. All Rights Reserved. Includes DVD creation technology used under license from Sonic Solutions. Copyright 1996-2005 Sonic Solutions. All Rights Reserved. This Product includes code licensed from RSA Data Security. Portions copyright Right Hemisphere, Inc. This product includes software developed by the OpenSymphony Group (http://www.opensymphony.com/).

Sorenson™ Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Notice to U.S. government end users. The software and documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250 ,and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Part Number: 90069163 (12/06)

Contents

Installing Flex Builder

This topic contains installation instructions for installing Adobe® Flex™ Builder™ 2 (both the stand-alone and plug-in configurations) and Adobe® Flash® Player 9 on Windows and Macintosh. Installation instructions for the entire Flex 2 product family can be found at http://www.adobe.com/go/flex2_installation.

Contents

Installing Flex Builder (stand-alone)

The stand-alone configuration of Flex Builder includes a customized version of Eclipse, Flex SDK 2, Flash Player 9, and its own Java Runtime Environment (JRE).

Read the release notes on the Adobe website at www.adobe.com/go/flex_documentation for the latest information or instructions.

To install the stand-alone configuration of Flex Builder:

1. Ensure that you are logged on as a user that has administrative privileges.
2. Flex Builder is delivered either on a DVD or as an online download. To begin the install process, insert the DVD and the Flex Builder installer will start automatically. If you have purchased Flex Builder as an online download, the installation process will start automatically when the file has finished downloading.

   In either case, if installation does not start automatically, you can use the Install Flex Builder 2.bat file (located on the DVD and in the folder in which the online download was extracted) to manually begin the installation process.
3. Select the “Flex Builder and Flex SDK” installation option.
4. Follow the prompts to begin the installation and to accept the license agreement.
5. Select the location where Flex Builder will be installed. By default, the location is as follows:
   - **Windows** C:\Program Files\Adobe\Flex Builder 2
   - **Macintosh** /Applications/Adobe Flex Builder 2
6. Select the Flash Player 9 browser plug-ins to install (on Macintosh, there is only one browser plug-in):
   - Internet Explorer
   - Netscape and Firefox
7. (Windows-only) When the installation is finished, restart your computer to ensure that the updated Flash Player browser plug-in is enabled.
8. (Windows Vista only) The first time you run Flex Builder 2, you must use the "Run as Admin", after which you can run it normally. To run as Admin, right click the Flex Builder 2 launch icon in the Start Menu, and select "Run as Admin".

Installing Flex Builder (plug-in)

The plug-in configuration of Flex Builder is an Eclipse plug-in that is installed into an existing version of Eclipse. In addition to the Flex Builder plug-in, it includes Flex SDK 2 and Flash Player 9. The plug-in configuration requires a minimum Eclipse version:

- **Windows** Eclipse 3.1.1
- **Macintosh** Eclipse 3.2

For the latest information or instructions, see the release notes on the Adobe website at www.adobe.com/go/flex_documentation.

To install the plug-in configuration of Flex Builder:

1. Ensure that you are logged on as a user that has administrative privileges.
2. Flex Builder is delivered either on a DVD or as an online download. To begin the install process, insert the DVD and the Flex Builder installer will start automatically. If you have purchased Flex Builder as an online download, the installation process will start automatically when the file has finished downloading.

   In either case, if installation does not start automatically, you can use the Install Flex Builder 2.bat file (located on the DVD and in the folder in which the online download was extracted) to manually begin the installation process.
3. Select the "Flex Builder Plug-In and Flex SDK" installation option.
4. Follow the prompts to begin the installation and to accept the license agreement.
5. Select the location where Flex Builder will be installed. By default, the location is as follows:
   - **Windows** C:\Program Files\Adobe\Flex Builder 2 Plug-in
   - **Macintosh** /Applications/Adobe Flex Builder 2 Plug-in
6. Select the location where Eclipse is installed (for example: C:\Eclipse on Windows or /Applications/eclipse on Macintosh).
7. Select the Flash Player 9 browser plug-ins to install (on Macintosh, there is only one browser plug-in):
   - Internet Explorer
   - Netscape and Firefox
8. (Windows-only) When the installation is finished, restart your computer to ensure that the updated Flash Player browser plug-in is enabled.

Installing Flash Player 9

You must use Flash Player 9 to run Flex 2 applications. Although the installer automatically installs Flash Player 9, you might need to install a different plug-in or run the stand-alone configuration.

To install Flash Player 9

1. Locate the installer you want.

   Flash Player 9 installers and the stand-alone Player are located in the following folder by default when the stand-alone configuration of Flex Builder is installed:

   Windows C:\Program Files\Adobe\Flex Builder 2\Player\debug

   Macintosh /Applications/Adobe Flex Builder 2/Player/debug

   The debug versions of the following players are included:

Installer	Version
Install Flash Player 9 AX.exe	Internet Explorer plug-in (ActiveX control)
Install Flash Player 9.exe	Mozilla, Firefox, or Netscape plug-in for Windows
Install Flash Player 9 UB.dmg	Macintosh plug-in (universal binary for PPC and Intel)
SAFlashPlayer.exe	Stand-alone Flash Player for Windows
standalone.app.hqx	Stand-alone Flash Player for Macintosh (automatically expanded during the Macintosh install process)

2. In Windows, run the .exe file.
3. For Macintosh, run the .dmg file.

About Flex Documentation

Getting Started with Flex 2 provides an introduction to the Adobe® Flex™ 2 product line, and contains a series of lessons designed to teach you the fundamentals of Flex. This manual is intended for application developers who are new to the Flex product line and require an overview of its features and capabilities.

Contents

Using this manual

This manual can help anyone get started with developing Flex applications. After reading this manual, you should read *Flex 2 Developer's Guide* for detailed information about Flex features and information about creating Flex applications, and the *Using Flex Builder 2* book for information on building applications using the Adobe® Flex™ Builder™ 2 IDE.

Getting Started with Flex 2 contains the following chapters:

Chapter	Description
Chapter 1, "Introducing Flex"	Introduces the Flex presentation server, a development and runtime environment that lets you create rich interfaces for your web applications.
Chapter 2, "Introducing Flex Builder 2"	Introduces Flex Builder, the integrated development environment (IDE) for developing applications using the Flex Framework.
Chapter 3, "Building a Flex Application"	Describes the basic process for building a Flex application.
Chapter 4, "Building a Flex Data Services Application"	Describes Flex Data Services.
Chapter 5, "Using Flex Charting Components"	Describes the Flex charting components.
Chapter 6, "Using MXML"	Describes MXML, the XML language for writing applications, and how developers use it to create applications.
Chapter 7, "Using ActionScript"	Introduces ActionScript and explains how to use ActionScript in an MXML application. Developers can use ActionScript to extend the functionality of their Flex applications.
Chapter 8, "Create Your First Application"	Shows you how to compile and run a Flex application in Flex Builder. Also Introduces you to the concept of projects in Adobe Flex Builder 2 and shows you how to create projects.
Chapter 9, "Retrieve and Display Data"	Shows you how to create a simple blog reader that retrieves recent posts and lets users read the first few lines of the posts.
Chapter 10, "Create a Constraint-based Layout"	Shows you how to create a constraint-based layout with Flex Builder.
Chapter 11, "Use List-based Form Controls"	Shows you how to populate list-based form controls with items to display and values to process.

Chapter	Description
Chapter 12, “Use an Event Listener”	Shows you how to use an event listener. It shows you how to write one for a Button control, and then how to call your event listener by using two different methods.
Chapter 13, “Use Behaviors”	Shows you how to add behaviors to a Flex user interface.
Chapter 14, “Use View States and Transitions”	Shows you how to use view states and transitions to create a user interface that reveals more information when users request it.
Chapter 15, “Create a Custom Component”	Shows you how to build an MXML component visually with Flex Builder. The lesson also shows you how to insert the new custom component visually in other MXML files.
Chapter 16, “Use the Code Editor”	Shows you the key code editing features in Flex Builder.
Chapter 17, “Debug an Application”	Shows you the basic steps of debugging your applications in Flex Builder.
Chapter 18, “Use Web Services”	Show you how to create a simple reporting application for a blog aggregator that lists the most popular posts in the last 30 days.
Chapter 19, “Use the Data Management Service”	Shows how to use the Data Management Service Java adapter for working with data that is persisted to a data store.
Chapter 20, “Use ColdFusion Event Gateway Adapter”	Shows you how to create a Flex application to send a message to a ColdFusion application.

Accessing the Flex documentation

The Flex documentation is designed to provide support for the complete spectrum of participants.

Documentation set

The Flex documentation set includes the following manuals:

Book	Description
Getting Started with Flex 2	Contains an overview of Flex features and application development procedures.
Using Flex Builder 2	Describes how to build an application using Flex Builder.
Flex 2 Developer's Guide	Describes how to develop your dynamic web applications.
Creating and Extending Flex 2 Components	Describes how to create custom components using MXML and ActionScript.
Building and Deploying Flex 2 Applications	Describes how to configure, develop, and deploy a Flex application.
Programming ActionScript 3.0	Describes how to use ActionScript.
Adobe Flex 2 Language Reference	Provides descriptions, syntax, usage, and code examples for the Flex API.

Viewing online documentation

All Flex documentation is available online in HTML and Adobe® Portable Document Format (PDF) files from the Adobe website. It is also available from the Adobe® Flex™ Builder™ Help menu.

Typographical conventions

The following typographical conventions are used in this book:

- *Italic font* indicates a value that should be replaced (for example, in a folder path).
- `Code font` indicates code.
- *`Code font italic`* indicates a parameter.
- **Boldface font** indicates a verbatim entry.

PART 1

Introducing Flex

1

This part contains an introduction to the the Adobe Flex 2 product line and Adobe Flex Builder.

The following chapters are included:

CHAPTER 1

Introducing Flex

1

This topic introduces you to the Adobe® Flex™ 2 product line. Flex delivers an integrated set of tools and technology that enable developers to build and deploy scalable rich Internet applications (RIAs). Flex provides a modern, standards-based language supporting common design patterns and includes a client runtime environment, a programming model, a development environment, and advanced data services.

This topic describes the characteristics of a typical Flex application and defines the Flex product line, the family of products that you use to build Flex applications.

Contents

About Flex

Flex encompasses both a standards-based programming model that will be familiar to professional developers and a set of products designed to enable efficient delivery of high-performance RIAs. RIAs combine the responsiveness and richness of desktop software with the broad reach of web applications to deliver a more effective user experience. Flex applications take advantage of Adobe® Flash® Player 9, enabling developers to seamlessly extend the capabilities of the browser and deliver richer, more responsive client-side applications as well as a more robust integration with server-side functionality and service-oriented architectures.

About the Flex product line

The Flex product line consists of a family of related products that let you design, develop, and deploy an entirely new class of RIA. The Flex product line includes the following products:

- Adobe® Flex™ 2 SDK (software development kit)
- Adobe® Flex™ Builder™ 2
- Adobe® Flex™ Data Services 2
- Adobe® Flex™ Charting 2

This section contains an overview of the Flex product line and of Flash Player.

About Flash Player 9

While Flex applications run in a browser, they take advantage of the Flash Player 9 runtime environment for executing client-side logic, rendering graphics, and playing animation, audio, or video. Flash Player 9 supports a new higher-performance version of ActionScript (ActionScript 3.0) that works in conjunction with the expressiveness innovations in Flash Player 8. Flash Player provides the platform for interface development so that both client and presentation-tier logic executes on the client computer. Because Flash Player runs consistently across all major operating systems and browsers, you do not have to program your applications for a specific browser or platform.

Additionally, Flex applications running in Flash Player 9 can interact with JavaScript logic or HTML content that is displayed in the browser. As a result, you can incorporate Flex into an existing web site or application, including working with other browser-based frameworks such as Ajax widgets or JavaScript libraries.

For more information, see "Deploying Flex applications on Flash Player" on page 25.

About Flex Software Development Kit 2

At the core of the Flex product line is the Flex Software Development Kit (SDK). Flex 2 SDK is the base set of technologies and utilities that you use to create applications by using the Flex product line. Flex 2 SDK consists of the Flex framework (class library), the Flex compilers, the debugger, the MXML and ActionScript programming languages, and other utilities. Flex SDK also includes the source code for the core Flex framework class library, enabling you to study the code of the classes as well as customize or extend them for your own use.

Flex SDK is available as a stand-alone package or as an integrated feature of Flex Builder and Flex Data Services.

For more information about building applications by using Flex 2 SDK, see "Flex deployment models" on page 25.

Flex applications consist of MXML and ActionScript source files:

- MXML is an XML language that you use to lay out the user interface for Flex applications. MXML provides tags that correspond to classes in the Flex framework and simplify usage for visual elements such as containers, navigators, and UI controls. You also use MXML to declaratively define nonvisual aspects of an application, such as access to network-based resources (for example, XML feeds and web services), server-based resources (such as the Flex Data Management Service feature of Flex Data Services), and data bindings between user-interface components and the data you retrieve. Flex SDK provides basic network access to web-based data, including remote access to web application servers, such as ColdFusion and PHP; Flex Data Services provides additional protocols and services. For more information, see Chapter 6, "Using MXML," on page 85.
- ActionScript is the programming language for Flash Player. ActionScript (which is based on ECMAScript and is similar to JavaScript) provides flow control and object manipulation features that are not available in MXML. For more information, see Chapter 7, "Using ActionScript," on page 91.

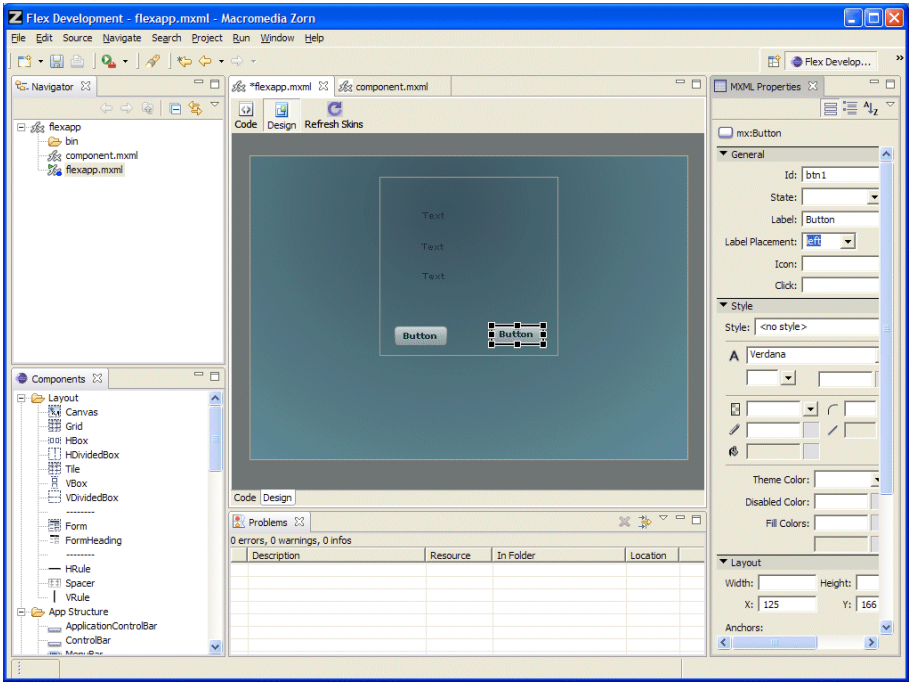
About Flex Builder 2

Flex Builder is an integrated development environment (IDE) for developing applications with Flex SDK, Flex Data Services, and the Flash Player. The Flex Builder IDE provides tools that help you develop, design, and debug Flex applications, including an integrated incremental compiler and a step-through debugger. Because it's tightly integrated with the Flex 2 SDK, the Flex application model, and the Flex programming languages, Flex Builder can improve the productivity of all members of your development team.

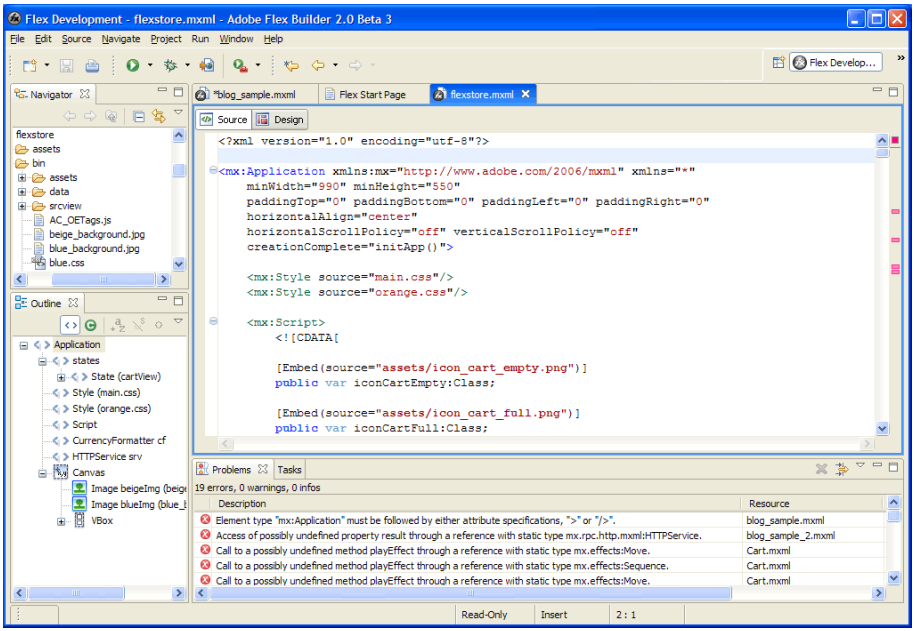
Flex Builder is built on the Eclipse workbench, an open source platform for building development tools. As a result, Flex Builder can be installed as a stand-alone product or as a set of plug-ins to an existing Eclipse installation, and can take advantage of hundreds of commercial and open source add-ons for the Eclipse workbench.

Flex Builder provides a set of code editors for working with MXML, ActionScript, and Cascading Style Sheets (CSS) as well as source code navigation tools to help you manage your code more easily, and a debugger to help you troubleshoot your applications. For user-interface design, Flex Builder supplies a visual design view, which allows developers or designers to lay out Flex components, customize their appearance, and design user interactivity.

The following example shows the Flex Builder interface in design view:



The following example shows the Flex Builder interface in code view:



About Flex Data Services 2

You build on the functionality of Flex 2 SDK by adding Flex Data Services. Flex Data Services adds enterprise messaging support and a greatly enhanced data services architecture to the Flex 2 SDK. You deploy Flex Data Services as a standard web application on your J2EE application server or servlet container. Flex Data Services simplifies the programming model for interacting with data on the server and includes the following features:

- A high-level programming model for synchronizing data changes between client and server or between multiple clients
- Integrated services for using publish-and-subscribe messaging
- Automated server data push and real-time data streaming
- An open adapter architecture for integrating with JMS, Hibernate, EJB, and other data persistence mechanisms.
- Authentication of client access to server resources
- Access to RemoteObjects by using the AMF protocol
- Data service logging

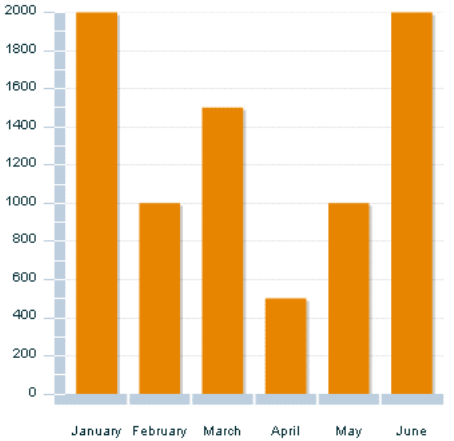
These features let you create and deploy enterprise-class applications that take full advantage of the rich presentation layer that the Flex 2 SDK provides. For more information, see "Data Access with Flex Data Services" on page 28.

About Flex Charting 2

The ability to display data in a chart or graph can make data interpretation much easier for application users. Rather than present a simple table of numeric data, you can display a bar, pie, line, or other type of chart using colors, captions, and a two-dimensional representation of your data.

Flex Charting components extend the Flex framework to add support for many of the most common chart types, including bar, pie, line, plot, and bubble. Charting components are dynamically rendered on the client computer, making it easy to add drill-down, rollover, and other interactivity that enhance the user experience. You can also use colors and captions to make your charts more readable.

A simple chart shows a single data series, where a series is a group of related data points. For example, a data series might be monthly sales revenues, or daily occupancy rates for a hotel. The following chart shows a single data series that corresponds to sales revenues for six months:



For more information on Flex Charting components, see Chapter 5, "Using Flex Charting Components," on page 77.

Benefits of using Flex

Enhanced user experience Flex lets you build applications that provide an engaging user experience. An engaging user experience ensures that customers are drawn into your application, that they understand how to use it, and that they can more quickly complete a task or find the information they are seeking.

A complete environment Flex is a powerful application development solution for creating and delivering RIAs within the enterprise and across the web. It provides a modern, standards-based language and programming model that supports common design patterns and includes a highly productive IDE.

Common deployment environment Flex applications execute on Flash Player 9, which is platform independent, so customers do not need to install custom client software. Also, Flash Player runs consistently in all browsers and platforms, so you do not have to worry about inconsistent behavior in different client environments.

Enterprise-class features You can use Flex Data Services to transparently synchronize data and support real-time data push. Messaging capabilities enable more robust applications that continue to function after network connectivity is lost and allow multiple people in different locations to browse or chat in the same application. These features, plus the ability to integrate audio and video, open the door to new ways of interacting with customers, partners, and employees.

Eliminate page loads Applications running in Flash Player behave like desktop applications, instead of a series of linked pages. Flash Player manages the client interface as a single, uninterrupted flow and does not require a page load from the server when the client moves from one section of the application to another.

Standards-based architecture Flex, ActionScript, and MXML are designed to existing standards. MXML is XML compliant, implements styles based on the Cascading Style Sheets, level 1(CSS1) specification, and implements an event model based on a subset of the W3C DOM Level 3 Events specification. ActionScript is an ECMAScript-based language that provides support for object-oriented development. The Flex server executes on standard J2EE platforms or servlet containers.

Cross-browser compatibility Web applications should run the same on all browsers and platforms. By standardizing on Flash Player as the client environment, you are guaranteed a consistent user experience on all platforms and browsers. For more information, see "Deploying Flex applications on Flash Player" on page 25.

Flex application characteristics

Many types of applications are appropriate for development in Flex. Some of the requirements of these applications, and how Flex supports these requirements, include the following:

Client data collecting Collecting user input is one of the most common uses for web applications. Flex supports forms, and all common form elements, to let you create rich and dynamic user experiences. Flex forms include hooks to the Flex data modeling and data validation mechanism, and the ability to identify required input fields. For more information, see Chapter 15, "Using Layout Containers," in the *Flex 2 Developer's Guide*.

Configuration One of the most common applications using Flex lets users perform product selection and configuration. The user works through a process to configure the features of a product, views or inspects the configuration, and then proceed through the steps required to complete a purchase.

Client-side processing of user input, including filtering and data validation Flex data management, which includes data models, data validators, data binding, and data services, lets you separate data representation from the way that a user views it. Typically, this design pattern is called Model-View-Controller (MVC). Flex also provides a powerful way to validate data and pass data between user-interface controls and external data sources with little or no server interaction. For more information, see Chapter 37, "Representing Data," in the *Flex 2 Developer's Guide*.

Direct user feedback Complex tasks must provide feedback to users when the user makes input errors or enters invalid information. Flex formatters and validators help ensure the quality of input data.

Multistep processes Many applications present the user with a process that includes a sequence of steps or decisions that require user input. For example, completing a registration form or checkout form often requires multiple steps to complete.

Ideally, you want your users to be able to navigate through multiple steps on a single page without losing the context of where they are in the process, and without losing any of the previous information that they have already entered. Flex supports the development of these applications by capturing state information, supporting browser Back and Forward buttons by using the History Manager. For more information, see Chapter 16, "Using Navigator Containers" and Chapter 32, "Using the History Manager," in the *Flex 2 Developer's Guide*.

Support for large data sets Enterprise data applications often handle large data sets that must be transmitted to the client from the server, or transmitted to the server from the client. These large data sets can overwhelm the bandwidth of your network, and lead to sluggish application performance. Flex Data Management Services, a feature of Flex Data Services, lets you break large data sets into smaller units so that no single client can monopolize the network.

Real-time data push Applications often share data among multiple clients. For example, an inventory management system must keep all clients synchronized on product availability. One way to implement this type of system is to have the clients poll the server at regular intervals to check for updates. However, this design uses unnecessary network bandwidth and processing cycles when no updates are available. Instead, Flex Data Services lets clients subscribe to data objects on the server. When the server updates a data object, it then pushes those updates out to all subscribing clients.

Occasionally connected clients Remote clients may not be able to maintain a connection to the server at all times. These clients are called *occasionally connected clients*. Flex Data Services let client applications perform offline data manipulation, and then automatically send updates to the server when the connection is restored.

Flex deployment models

The Flex platform supports a range of deployment models:

Client-side only Applications run on the client and use no server resources.

Access server data through simple RPC services Applications interact with remote data through web services and HTTP services.

Flex Data Services Applications use the enhanced data architecture of Flex Data Services to provide advanced features, such as data synchronization, security, and messaging.

These models are discussed in the following sections, following a discussion of how you use Flash Player with Flex.

Deploying Flex applications on Flash Player

The target of a Flex application is Flash Player. Flex supports two versions of Flash Player. Flex application users install the *standard* Flash Player to run Flex applications. Flex also supports the *debug* version of Flash Player, called Flash Debug Player. Flex application developers use Flash Debug Player during the development process. Flash Debug Player, which is installed by default with Flex Builder and Flex Data Services, supports important features for developers, including the following:

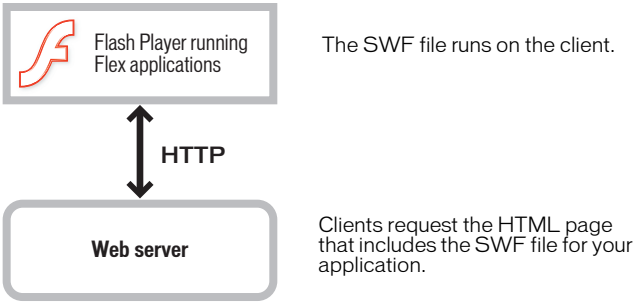
Error reporting Lets you direct runtime error and warning messages to a log file. Flash Debug Player can also capture the output of the `trace()` function and write it to the log file. For more information, see Chapter 11, "Running and Debugging Applications," in *Using Flex Builder 2*, and Chapter 11, "Logging," in *Building and Deploying Flex 2 Applications*.

Debugging support Lets you debug ActionScript files that your Flex applications use. For more information, see Chapter 11, "Running and Debugging Applications," in *Using Flex Builder 2*, and Chapter 12, "Using the Command-Line Debugger," in *Building and Deploying Flex 2 Applications*.

Client-side only applications

You typically deploy Flex applications as SWF files embedded in HTML, ColdFusion, PHP, or other types of web pages. Users run your Flex application by requesting the associated web page, which then downloads the SWF file to your browser for execution by Flash Player. Although most Flex applications provide some level of server interaction, you can write Flex applications (a configurator, for example) that provide stand-alone functionality when running on the client.

The following example shows an application that uses Flex SDK to deploy a client-side only application:



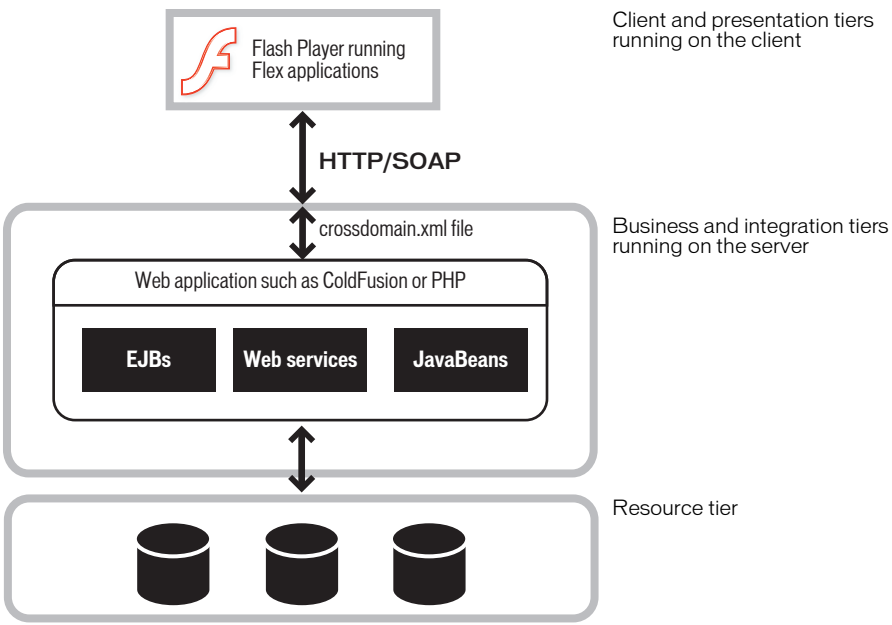
Data access with HTTPService and WebService

The Flex HTTPService and WebService tags let you retrieve data from a remote server. These tags, which are also called remote procedure call (RPC) components, let your application interact with remote servers to provide data to your applications, or for your application to send data to a server. Using Flex 2 SDK, you can build applications that access remote data from a web application server, such as ColdFusion or PHP, through SOAP (web services), or HTTP GET or POST requests (HTTP services). The web application retrieves the appropriate data (typically from a database, for example), formats it (typically as XML), and returns it to the client application.

For security, by default Flash Player does not allow an application to access a remote data source from a domain other than the domain from which the application was served. Therefore, a remote server must either be in the same domain as the server hosting your application, or the remote server must define a *crossdomain.xml* file.

A crossdomain.xml file is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from certain domains, or from all domains. The crossdomain.xml file must be in the web root of the server that the Flex application is contacting.

The following example shows an application that uses Flex SDK to access data from a server by using RPC services (the originating web server is omitted for clarity):



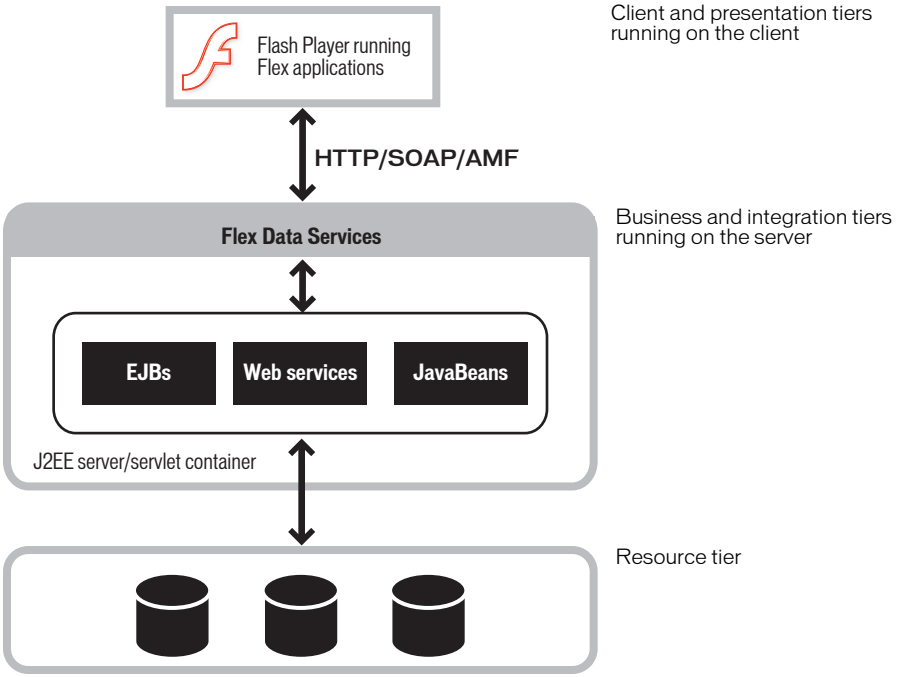
With Flex 2 SDK, Flash Player provides the platform for interface development so that both client and presentation tier logic executes on the client computer. Flex 2 SDK applications that execute in Flash Player process user interactions; perform data validation; issue HTTP and SOAP requests; and perform other operations previously performed on the server. Because Flash Player runs consistently across all major operating systems and browsers, you do not have to program your applications for a specific browser or platform.

NOTE	You can still develop presentation-tier logic for the server as part of your application, and connect to that logic from the client.

Data Access with Flex Data Services

When you use Flex Data Services to develop applications, you take advantage of its enhanced data services architecture. For more information, see Chapter 4, "Building a Flex Data Services Application," on page 71.

The following example shows the possible tiers for a Flex application implemented by using Flex Data Services:



This example shows Flex Data Services executing on the application server. Flex applications executing in Flash Player can pass data to and receive data from Flex Data Services.

Using the enhanced data services architecture

Flex Data Services provide a data services architecture that greatly enhances the services of the Flex 2 SDK. These enhanced data services add support for many features important to enterprise application development, including the following:

Transparent data synchronization Lets you keep multiple clients synchronized. Multiple Flex clients subscribe to a single server-side data object. Flex Data Services then broadcast updates to the clients to publish any changes to it, meaning your client applications do not have to poll the server to check for updates.

Publish and subscribe data services Let multiple client applications communicate with each other. For example, you can implement an instant messaging service among multiple clients.

JMS messaging support Java Message Service (JMS) is a Java API that lets applications send data to, and receive data from a Java application.

Client authentication Lets you secure your data services by using basic or custom authentication.

The data services architecture is based on an underlying messaging infrastructure. Flex Data Services use XML configuration files to manage message topics and queues on the server. As part of configuring messaging, you can set the maximum message size. If a message exceeds this maximum value, multiple message batches are used, where the separate messages are reassembled by the client. This enables asynchronous data paging across the network.

Additional RPC services with Flex Data Services

The RPC services of Flex Data Services are enhanced to add support for using the AMF protocol to access RemoteObjects. This lets you access Java objects (JavaBeans, EJBs, POJOs) on remote servers.

Flex Data Services includes a proxy that can intercept RPC requests from Flex applications to remote servers, redirect the requests to the specified server, and then return the response to the client. Because the proxy acts as the broker for remote server access, you are not required to define a crossdomain.xml file on the remote servers, as you are when you use the Flex 2 SDK. However, you can still access remote servers directly if they have defined a crossdomain.xml file.

Summary of Flex application features

The following table describes the features that you are most likely to use when building Flex applications:

Feature	Description
User-interface controls	Controls are user-interface components, such as Button, TextArea, and ComboBox controls. You use MXML tags to add controls to an application. For more information, see Chapter 9, "Using Controls," in the *Flex 2 Developer's Guide*.
User-interface containers	Containers are user-interface components that let you control the layout characteristics of the user-interface components that they contain. You can use containers to control child sizing and positioning, or to control navigation among multiple child containers. You use MXML tags to add containers to an application. For more information, see Chapter 13, "Introducing Containers," in the *Flex 2 Developer's Guide*.
MXML components	MXML components are Flex components written in MXML files. They provide an easy way to extend an existing Flex component and encapsulate the appearance and behavior of a component in a custom MXML tag. You use MXML tags to add MXML components to an application. For more information, see Chapter 2, "Creating Flex Components," in *Creating and Extending Flex 2 Components*.
ActionScript components	ActionScript components are Flex components written in ActionScript classes. They are a good choice for nonvisual components. You can use MXML tags to add ActionScript components to an application. For more information, see Chapter 2, "Creating Flex Components," in *Creating and Extending Flex 2 Components*.
Data binding	The data binding feature provides a simple syntax for automatically copying the value of a property of one client-side object to a property of another object at runtime. For more information, see Chapter 39, "Storing Data," in the *Flex 2 Developer's Guide*.

Feature	Description
Data services	Data service objects let you interact with server-side data sources. You can work with data sources that are accessible by using SOAP-compliant web services, Java objects, or HTTP GET or POST requests, and RemoteObjects accessed by using the AMF protocol if you have the Flex Data Services. For more information, see Chapter 39, "Storing Data," in the *Flex 2 Developer's Guide*.
Data validation	Data validators help you ensure that the values in the fields of a data model meet certain criteria. For example, you can use a validator to check whether a user entered a valid ZIP code value in a TextInput control. For more information, see Chapter 40, "Validating Data," in the *Flex 2 Developer's Guide*.
Data formatting	Data formatters let you format data into strings before displaying it in the user interface. For example, you can use a formatter to display a phone number in a specific format. For more information, see Chapter 41, "Formatting Data," in the *Flex 2 Developer's Guide*.
History management	History management lets users navigate through a Flex application by using the web browser's Back and Forward navigation commands. It also lets users return to the previous location in an application if the browser accidentally navigates away from that location. For more information, see Chapter 32, "Using the History Manager," in the *Flex 2 Developer's Guide*.
Drag-and-drop management	Drag-and-drop management lets you move data from one place in a Flex application to another. This feature is especially useful in a visual application where your data can be items in a list, images, or Flex components. For more information, see Chapter 29, "Using the Drag and Drop Manager," in the *Flex 2 Developer's Guide*.
Styles, fonts, and themes	Styles, fonts, and themes help you define the overall appearance of applications. You can use them to change the appearance of a single component, or apply them across all components. For more information, see Chapter 18, "Using Styles and Themes," in the *Flex 2 Developer's Guide*.
Behaviors	Behaviors let you add animation or sound to applications in response to user or programmatic action. For more information, see Chapter 17, "Using Behaviors," in the *Flex 2 Developer's Guide*.

Feature	Description
Repeaters	Repeaters let you dynamically repeat any number of controls or containers specified in MXML tags, at runtime. For more information, see Chapter 26, "Dynamically Repeating Controls and Containers," in the *Flex 2 Developer's Guide*.
Image and media importing	You can use MXML tags to import several images into applications. Flex supports several formats, including JPEG, PNG, GIF, and SVG images and SWF files. In addition, you can use the VideoDisplay control to incorporate streaming media into Flex applications. Flex supports the Flash Video File (FLV) file format with this control. For more information, see Chapter 30, "Embedding Assets," in the *Flex 2 Developer's Guide*.
View states	View states let you structure your application to present a varying appearance by defining a base application view state, and sets of changes that modify the base view state. Each view state can add or remove children, set or change styles and properties, or define state-specific event handlers. You can also define transitions between view states, which control the appearance of the change from one view state to another. For more information, see Chapter 27, "Using View States," in the *Flex 2 Developer's Guide*.
ActionScript scripting	ActionScript lets you perform actions with the components that are represented by MXML tags. You use ActionScript in your Flex applications to do the following: • Handle events • Define custom functions and methods • Call ActionScript functions • Work with components after they are instantiated For more information, see Chapter 9, "Code Editing in Flex Builder," in *Using Flex Builder 2*, or Chapter 4, "Using ActionScript," in the *Flex 2 Developer's Guide*.
Debugging	Flex includes support for debugging and warning messages, an error-reporting mechanism, and a command-line ActionScript debugger to assist you in debugging your application. For more information, see Chapter 11, "Running and Debugging Applications," in *Using Flex Builder 2*, or Chapter 12, "Using the Command-Line Debugger," in *Building and Deploying Flex 2 Applications*.

Where to next

This book contains an introduction to Flex and an overview of developing Flex applications. The Flex documentation set contains detailed information about these topics. For more information, see the following:

- For information on MXML and ActionScript, see Chapter 6, "Using MXML," on page 85, and Chapter 7, "Using ActionScript," on page 91.
- For information on using Flex Builder, see *Using Flex Builder 2*.
- For information on using Flex components, see the *Flex 2 Developer's Guide*.
- For information on developing custom components, see *Creating and Extending Flex 2 Components*.
- For more information on the Flex data model, see the *Flex 2 Developer's Guide*.
- For more information on configuring, developing, and deploying Flex applications, see *Building and Deploying Flex 2 Applications*.
- For information on debugging, see *Using Flex Builder 2* and the *Flex 2 Developer's Guide*.

CHAPTER 2

Introducing Flex Builder 2

2

This chapter introduces you to Adobe® Flex™ Builder™ 2. An integrated development environment (IDE) for developing applications using the Adobe® Flex™ 2 framework and the Flash API, Flex Builder provides you with the tools you need to develop, design, and debug Flex 2 and ActionScript 3.0 applications.

Contents

About Flex Builder

Built upon the Eclipse workbench (an open-source IDE), you use Flex Builder to develop Flex 2 and ActionScript 3.0 applications using coding, design, and debugging tools.

Flex Builder and Eclipse

Eclipse is an open-source integrated development environment (IDE), which can be extended using custom plug-ins. Flex Builder leverages the Eclipse framework to provide an IDE that allows you to develop Flex 2 and ActionScript 3.0 applications, as well as Flex libraries. The underlying development workbench builds on the standard Eclipse workbench features, so Eclipse users will find Flex Builder familiar and intuitive. For more information, see “Flex Builder basics” in *Using Flex Builder 2*.

Flex Builder versions and configurations

Flex Builder is available in two versions: the standard version and a version that includes the Flex Charting components. (For more information, see "Flex Builder versions" in *Using Flex Builder 2*.)

Each version is available in two configurations: standalone and plug-in. The standalone configuration is a customized packaging of Eclipse and the Flex Builder plug-ins created specifically for developing Flex and ActionScript applications. The plug-in configuration is for users who already use the Eclipse workbench, who want to add the Flex Builder plug-ins to their toolkit of Eclipse plug-ins. (For more information, see "Flex Builder configurations" in *Using Flex Builder 2*.

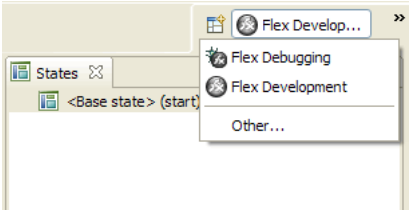
Flex Builder and connecting to data

Since Flex applications do not directly connect to a database, you use other tools and services such as PHP, ASP, JSP, Web services, Flex Data Services, ColdFusion Flash Remoting Services, and so on. Flex Builder allows you to create projects that use these various tools and services to access data. You then directly interact with them in your MXML and ActionScript code. For more information about creating projects and accessing data, see Chapter 3, "Working with Projects" in *Using Flex Builder 2*.

About Flex Builder perspectives

A group of editors and views that support a specific task or group of tasks are combined into a *perspective*. Flex Builder contains two perspectives: one for general development and design and the other for debugging.

Perspectives change automatically to support the task at hand. For example, when you create a Flex project, the workbench switches to the Flex Development perspective; when you start a debugging session, the Flex Builder debugging perspective is displayed when the first breakpoint is encountered. You can also manually switch perspectives yourself by selecting Window > Open Perspective from the main menu. Or, you can use the *perspective bar*, which is located in the workbench tool bar.

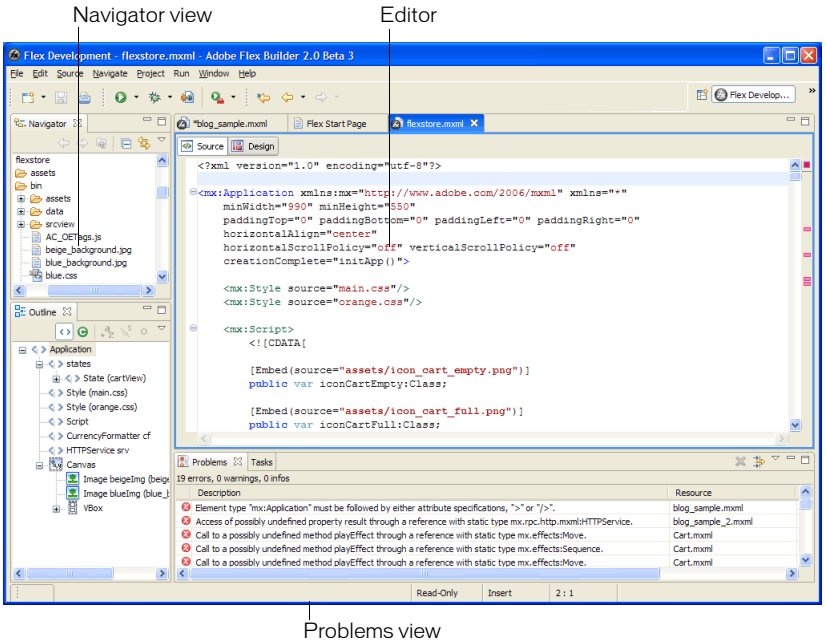


If you're using the plug-in configuration of Flex Builder and have other Eclipse plug-ins installed, you may have many additional perspectives. While perspectives are delivered with each plug-in, you may customize them to your liking or create your own. Customizing or creating a perspective is a simple matter of selecting, placing, and sizing the editors and views you need to accomplish your development tasks. For more information about working with and customizing perspectives, see Chapter 4, "Navigating and Customizing the Flex Builder Workbench" in *Using Flex Builder 2*.

The Flex Development perspective

You use the Flex Builder code editor to write your Flex and ActionScript applications. When creating Flex applications you start with the main application MXML file. You can then embed ActionScript and CSS code into the MXML file or create separate ActionScript or CSS files and instead import them into MXML files.

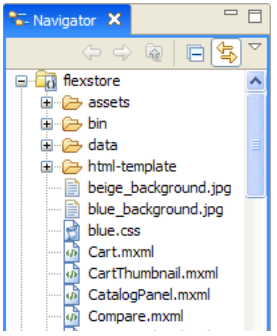
The code editor is contained within the Flex Development perspective, which also includes the supporting Navigator, Problems, and Outline views. When you create a project, Flex Builder switches into the development perspective so you can begin developing your application.



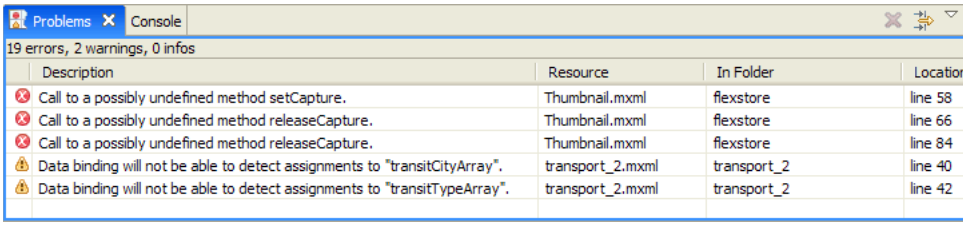
The Flex Development perspective contains the following elements:

Code editor Allows you to write MXML, ActionScript, and CSS code. Provides code hinting and formatting, a design mode where you can visually define your Flex application layout, complete integration with the Flex Builder debugging tools, and syntax error checking and problem reporting. For more information about using the code editor, see Chapter 9, "Code Editing in Flex Builder" in *Using Flex Builder 2*.

Navigator view Displays all of the projects in the workspace and all of the resources (folders and files) within your projects. For more information about the Navigator view and working with projects, "Managing project resources" in *Using Flex Builder 2*.



Problems view Displays syntax and other compilation errors that are detected by the compiler. Since by default the compiler builds your Flex project each time you make a change to it, you get nearly immediate feedback as you write code. For more information about working with the Problems view, see "Using the Problems view" in *Using Flex Builder 2*.



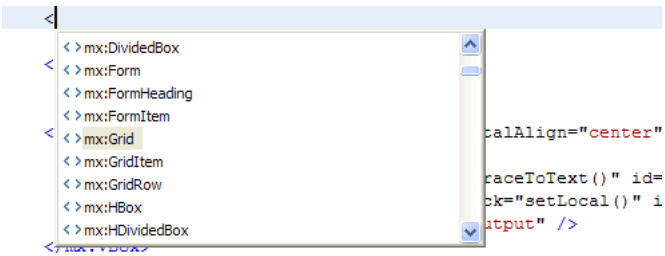
NOTE

You can also optionally add the Tasks and Bookmarks views. These views provide additional shortcuts for managing and navigating your code. For more information about these views, see "About markers" in *Using Flex Builder 2*.

Code Assistance in Flex Builder

The code editor contains many features that simplify and streamline code development. Foremost among these features is Content Assist, which displays code completion hints as you enter MXML, ActionScript, and CSS code into the editor.

Code hints appear automatically as you enter your code. You can also display code hints by pressing Control+Space.



Code hints appear whenever the Flex or language (MXML, ActionScript, and CSS) provides options for you to complete the current code expression. For example, if you type the name of a Flex component, you are prompted with a list of all properties of that component.

ActionScript code hinting is also supported. ActionScript code hints are displayed within embedded `<mx:Script>` tags in an MXML document and within stand-alone ActionScript files in your project. Content Assist hints all ActionScript language elements: interfaces, classes, variables, functions, return types, and so on.

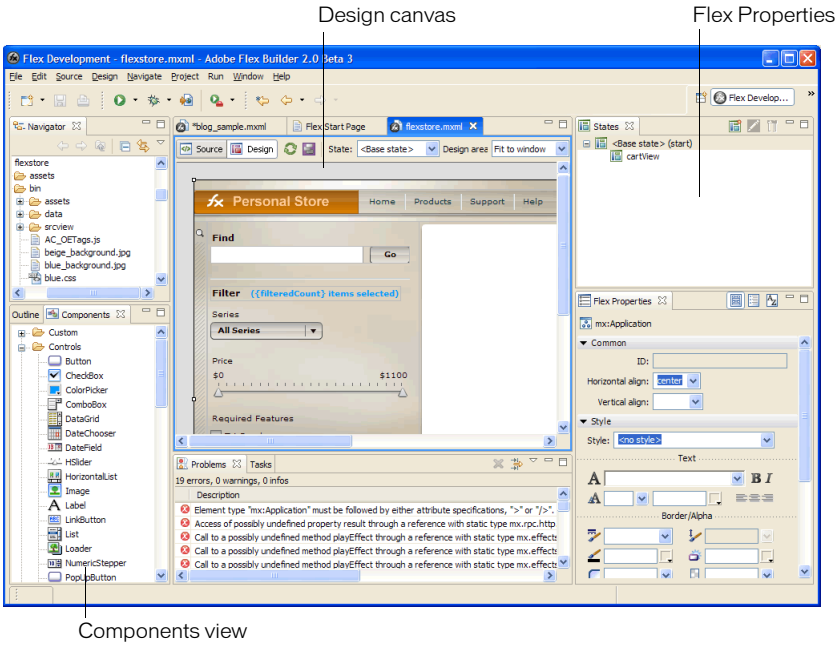
Content Assist also provides hinting for any custom MXML components or ActionScript classes that you create yourself and which are part of your project. For example, if you define a custom MXML component and add it to your project, code hints are displayed when you refer to the component in your MXML application.

Designing Flex applications in Flex Builder

The development perspective has two modes: one for writing code and the other for visually laying out your application. You can switch between the two modes by selecting either the Code or Design buttons from the code editor toolbar. When you switch into Design mode, the development perspective displays the design canvas and the supporting Components and Flex Properties views.

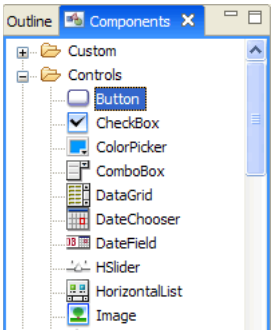
NOTE

Design mode is not available when working with ActionScript applications. To preview your ActionScript applications, you need to build and run them. For more information, see Chapter 10, "Building Projects" and Chapter 11, "Running and Debugging Applications" in *Using Flex Builder 2*.

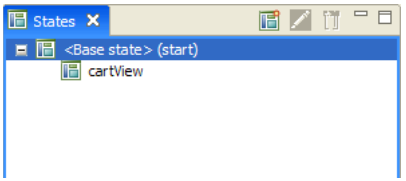


Design canvas While in the code editor, you can switch into Design mode and then drag containers and components onto the design canvas, size and move containers and components, and otherwise visually interact with your Flex application.

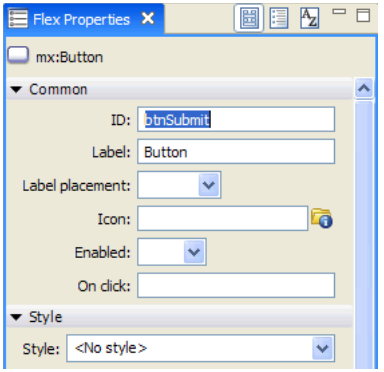
Components view All of the Flex containers and components are contained in this view and you may select and add them to the design canvas. As you create your own custom components, they will also be displayed in the Components view.



States view You can use Flex Builder to create applications that change their appearance based on the task the user is performing. These user interface transformations are referred to as *view states*. You create and manage view states in the States view. For more information about view states, see Chapter 6, "Adding View States and Transitions" in *Using Flex Builder 2*.

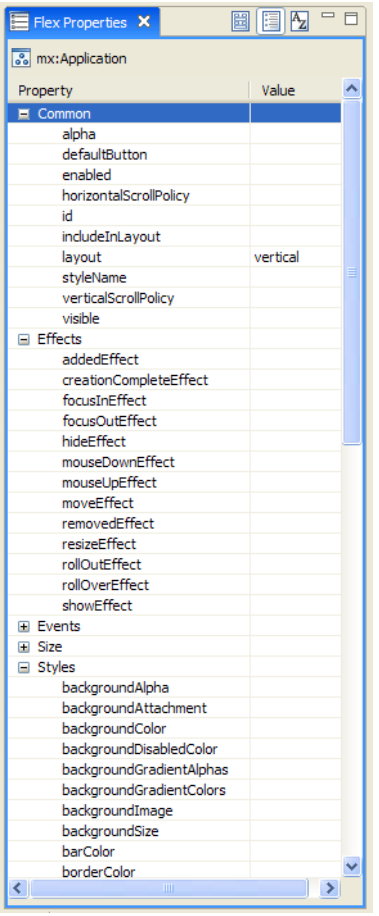


Flex Properties view When a container or component is selected its properties are displayed in the Flex Properties view. You may set and edit properties as appropriate.

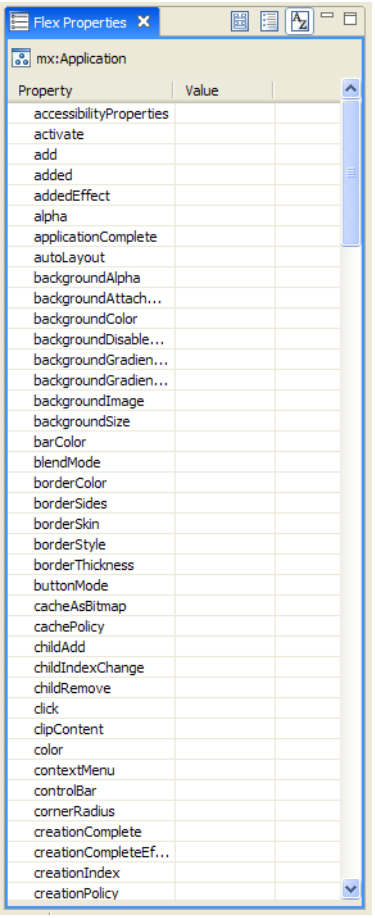


The Standard View shows some of the most commonly used general properties for that control or container (such as ID). It also shows common style properties and common layout properties.

You must change the properties view to see a list of all the available properties for a container or component. You can select from one of the view buttons to change the properties view. The default selection is the Standard View (indicated by the Standard View command). If you select the Category View command (), Flex Builder displays a list of properties, which are organized by category (such as Common, Effects, Events, and Styles).



If you select the Alphabetical View command ( ), Flex Builder displays a complete alphabetical list of the container's or component's properties.

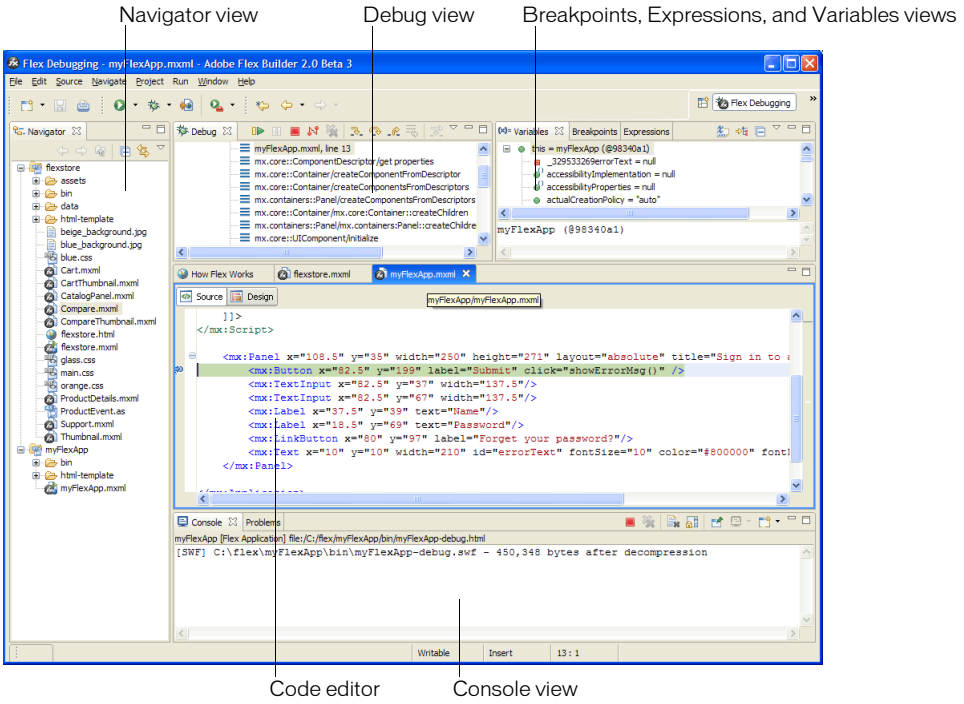


For more information about designing Flex applications in Flex Builder, see Chapter 5, "Building a Flex User Interface" in *Using Flex Builder 2*.

The Flex Debugging perspective

The Flex Debugging perspective contains the tools you need to debug your applications. Like the development perspective, the primary tool within the debugging perspective is the code editor. In the context of debugging your applications, the code editor is used to locate and highlight lines of code that need attention so that you can fix them and continue testing your application.

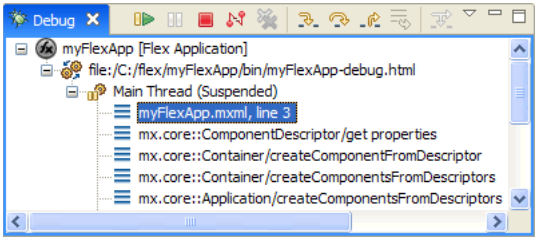
For example, you can set breakpoints in your script to stop the execution of the script so that you can inspect the values of variables and other information up to that point. You can also step to the next breakpoint or step into a function call to see the variable values change.



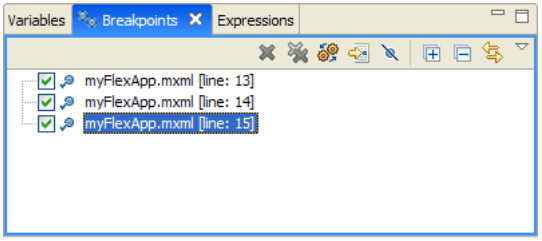
The Flex Debugging perspective is displayed automatically when you begin a debug session. You can also switch to it yourself by selecting it from the perspective bar, which is located at the right edge of the main toolbar.

The Flex Debugging perspective contains the following views:

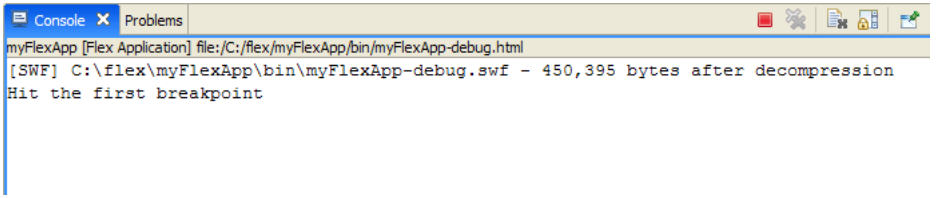
Debug view The Debug view (in other debuggers this is sometimes referred to as the *callstack*) displays the stack frame of the suspended thread of the Flex application you are debugging. You use the Debug view to manage the debugging process. For example, the Debug view allows you to resume or suspend the thread, step into and over code statements, and so on.



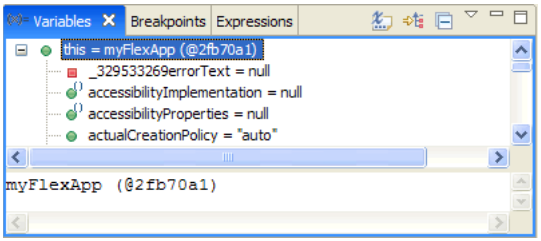
Breakpoints view The Breakpoints view lists all the breakpoints you set in your project. You can double-click a breakpoint and display its location in the code editor. You can also disable, skip, and remove breakpoints.



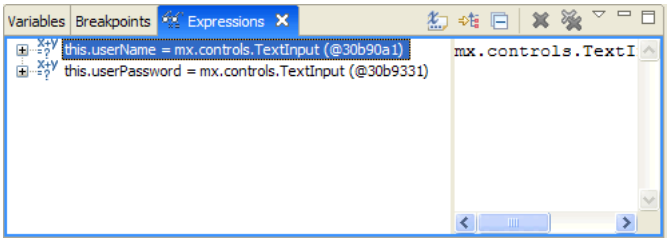
Console view The Console view displays the output from trace statements placed in your ActionScript code.



Variables view The Variables view serves the same purpose as the *locals* window does in other debuggers. It displays information about the variables in the currently-selected stack frame.



Expressions view The Expressions view serves the same purpose as the *watch* window does in other debuggers. It is used to monitor a set of critical variables. You can choose the variables you consider critical in the Variables view and add them to and monitor (watch) them in the Expressions view.

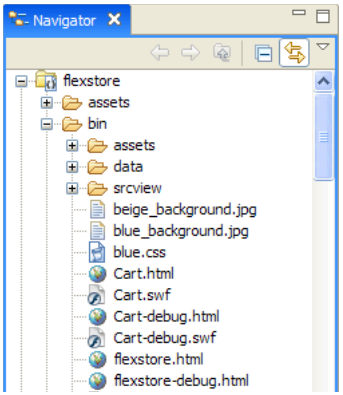


When you debug your application you can then monitor and, if needed, modify the values. You may also add and remove variables within the Expressions view.

For more information about debugging Flex and ActionScript applications, see Chapter 11, "Running and Debugging Applications" in *Using Flex Builder 2*.

Compiling your applications

Flex Builder automatically compiles your Flex and ActionScript 3.0 projects into application SWF files whenever changes are made to project resources. For example, when you add new a new file to a project, the project is compiled. You can override this default and instead compile your applications manually, whenever you choose. When your projects are compiled, automatically or manually, a release and debug version of your application SWF files are placed in the project output folder along with the HTML wrapper files, and so on.



Once your projects are compiled into application SWF files, you run and debug them as needed. For more information about running and debugging your applications, see Chapter 11, “Running and Debugging Applications” in *Using Flex Builder 2*.

Running and debugging your applications

After your projects are built, you may run and debug them as applications to test their functionality. Running and debugging your projects opens the main application SWF file in your default web browser or directly in Flash Player.

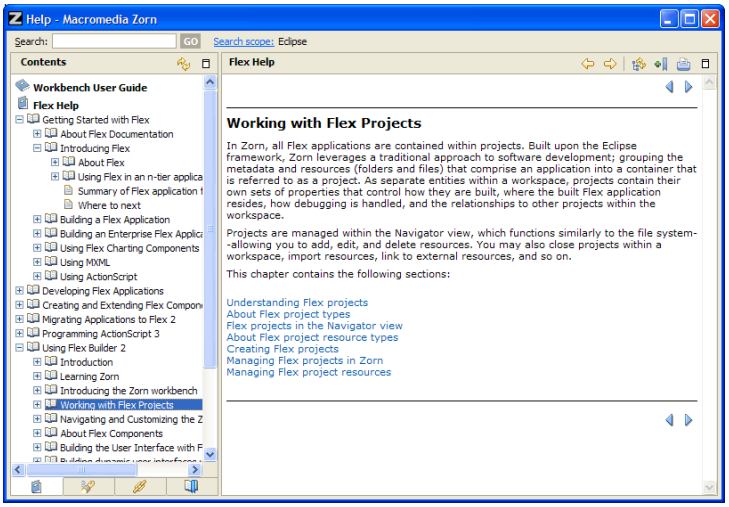
Your applications run based on a launch configuration. When you create new Flex and ActionScript applications, a launch configuration specifies the location of the built applications files, the main application file, and so on. In most cases the default launch configuration is all you’ll need to run and debug your applications. You can, however, modify the launch configuration or create custom launch configurations.

For more information, see Chapter 11, “Running and Debugging Applications” in *Using Flex Builder 2*.

More information about Flex Builder

This chapter has presented a brief introduction to Flex Builder. Many of the Flex Builder features were skimmed over or excluded. For a much more thorough explanation of the features and uses of Flex Builder refer to the following documentation:

Using Flex Builder 2 This user’s guide is available in the Flex Builder help system and in PDF format. To access the Help system, select Help > Help Contents. All of the Flex product line documentation is available, including *Using Flex Builder 2*.



You can also access context-sensitive help anywhere in Flex Builder by pressing F1.

Flex Builder Getting Started Lessons These simple lessons will help you quickly learn the basics of Flex and Flex Builder. They are available in the Flex Builder help system and in PDF format. For more information, see “Lessons” on page 99 of this guide.

For more information about using the Flex Builder Help system, see “Using the Flex Builder help system” in *Using Flex Builder 2*.

PART 2

Flex Basics

2

This part presents an overview of the products and technologies that comprise the Flex environment.

The following chapters are included:

CHAPTER 3

Building a Flex Application

3

Flex defines a component-based development model that you use to build your applications. To effectively design and build your applications, you should be familiar with this model, and with the steps of the application development and deployment process.

This chapter describes the development process that you use to create an applications. Included in this chapter is information on the decisions that you have to make to tailor an application to your specific requirements, overviews of the development tools supplied with the Flex product line, and comparisons to other technologies such as HTML and ColdFusion.

Contents

Developing applications

Flex supports a component-based development model. You can use the prebuilt components included with Flex, you can extend the Flex component library by creating new components, or you can combine prebuilt components to create composite components.

Application model

When you build an application using Flex, you describe its user interface using components called containers and controls. A *container* is a rectangular region of the screen that contains controls and other containers. Examples of containers are a Form container used for data entry, a Box, and a Grid. A *control* is a form element, such as a Button or Text Input field.

For example, the following figure shows two different Box containers that each contain three Button controls and a ComboBox control:

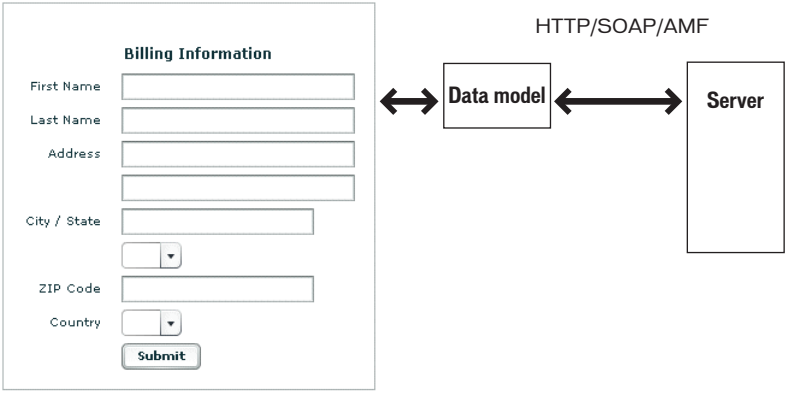
Box container with horizontal layout

Box container with vertical layout

This figure shows the controls within a horizontal Box (HBox) container. An HBox container arranges its controls horizontally across the Flash Player drawing surface. The figure also shows the controls in a vertical Box (VBox) container. A VBox container arranges its controls vertically.

Containers and controls define the application's user interface. In an MVC design pattern, those pieces of the application model represent the *view*. The *model* is represented by the data model. Flex data models let you separate your application's data and business logic from the user interface.

You define your data models using MXML or ActionScript as part of a Flex application. The following figure shows a form created in Flex that uses a data model:



Data binding is the process of tying the data in one object to another object. The data model supports bidirectional data binding for writing data from Flex controls to the data model, or for reading data into controls from the model. You can also bind server data to a data model or directly to Flex controls. For example, you can bind the results returned from a web service to the data model, and then have that data propagate to your form controls.

The data model supports automatic data validation. This means that you can use the Flex validators, such as the ZipCode validator, to ensure that the value in a model field is a valid ZIP code. If the data is invalid, you can display a message to the user so that the user can correct the error.

How Flex fits the MVC models

The goal of the Model-View-Controller (MVC) architecture is that by creating components with a well-defined and limited scope in your application, you increase the reusability of the components and improve the maintainability of the overall system. Using the MVC architecture, you can partition your system into three categories of components:

Model components Encapsulates data and behaviors related to the data.

View components Defines your application's user interface.

Controller components Handles the data interconnectivity in your application.

For example, with the MVC design, you could implement a data-entry form that has three distinct pieces:

- The model consists of XML data files or the remote data service calls to hold the form data.
- The view is the presentation of any data and display of all user interface elements.
- The controller holds the user interface logic.

Although you can consider a Flex application as part of the View in a distributed MVC architecture, you can use Flex to implement the entire MVC architecture on the client. A Flex application has its own view components that define the user interface, model components to represent data, and controller components responsible for the communication with back-end systems. This client-side separation of task is not available in HTML.

Working with a web server

Your development and deployment environments typically include a web server, where you use the web server to return a Flex SWF file in response to a user request. You use one of the following types of web servers:

- Simple web server

  A simple web server only responds to static page requests to simple HTML pages. In this case, you precompile your Flex applications and write a wrapper that embeds your Flex application's SWF file in an HTML page.

- Web application server

  A web application server, such as JRun, ColdFusion, or PHP, can dynamically generate pages that host your Flex applications. In this case, you can take advantage of the application server's available tag libraries and processing language to dynamically create a wrapper for your Flex application. However, you must precompile your Flex application before deploying it to this type of server. You can use any type of server, and not just a Java application server, to serve up Flex applications, as long as you precompile that application and that application does not use the services available with Flex Data Services.
- J2EE application server or servlet container

  You require a J2EE application server or servlet container, such as JRun, Tomcat, or WebSphere, to run Flex Data Services. You typically precompile your Flex applications before deploying them on a server running Flex Data Services. However, during development you can request a Flex application's MXML file in the browser to compile the Flex application at run time. This invokes the web-tier compiler which generates a wrapper and returns a compiled SWF file.

  You can optionally install the integrated JRun J2EE server when you install Flex Data Services. The integrated JRun server is a development-only version of the JRun 4 application server that is not intended for deployment. The integrated version of JRun also includes the JRun Web Server (JWS) that you can use to handle HTTP requests. This web server is also not intended for deployment.

At a minimum, you should have a simple web server in your development environment. Without a web server, you can only run a Flex application in the standalone Flash Player or request a SWF file directly in your browser. The former technique is not recommended because it prevents your application from using many networking features of Flex. The latter technique is not recommended because not all browsers support direct SWF-file requests.

Typical application development steps

You typically develop a Flex application by using the following steps:

1. Within a text editor or integrated development environment (IDE), such as Adobe Flex Builder, Eclipse, or IntelliJ, insert the MXML root tags into an MXML file.
2. Add one or more containers.
3. Add controls to a container, such as input fields, buttons, and output fields.
4. Define a data model.
5. Add a web service, HTTP service, or request to a remote Java object.
6. Add validation to input data.

7. Add a script to extend a component.
8. Compile your application into a SWF file.

NOTE

If you have Flex Data Services, you can deploy your application as a set of MXML and ActionScript files. Upon receiving an HTTP request to an MXML file, Flex Data Services compiles your application into a SWF file. For more information, see Chapter 4, "Building a Flex Data Services Application," on page 71.

Deploying applications

You can deploy your application as a compiled SWF file or, if you have Flex Data Services, you can deploy your application as a set of MXML and ActionScript files.

Deploying a SWF file

After you compile your application into a SWF file, you deploy it by copying it to a directory on your web server or application server. Users then access the deployed SWF file by making an HTTP request in the form:

```
http://hostname/path/filename.swf
```

Although you can directly request a SWF file, you typically incorporate the SWF file into a web page by using a wrapper. The *wrapper* is responsible for embedding the Flex application's SWF file in a web page, such as an HTML, ASP, JSP, or ColdFusion page. In addition, you use the logic in the wrapper to enable history management, Express Install, and to ensure that users with and without JavaScript enabled in their browsers can access your Flex applications.

For more information on wrappers, see Chapter 16, "Creating a Wrapper," in *Building and Deploying Flex 2 Applications*.

Deploying MXML and ActionScript files

If you have Flex Data Services, you can deploy your application as a set of MXML and ActionScript files. When you deploy your application as a set of MXML and ActionScript files, the user requests the main MXML file to start the application. The first time a user requests the MXML file URL in a web browser, the server compiles the MXML code into a SWF file. The server then sends the SWF file to the web browser where it is rendered in Flash Player.

Flex application files use the MXML filename extension. You store these files under the web root directory of your J2EE application.

To request an application deployed as MXML and ActionScript files, the user makes a request to the main MXML file in the form:

```
http://hostname/path/filename.mxml
```

The main MXML file contains the `<mx:Application>` tag. For more information, see Chapter 6, "Using MXML," on page 85.

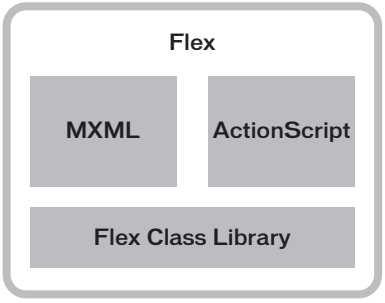
Upon receiving an HTTP request for an MXML file, Flex performs the following steps:

1. Compiles the MXML file to produce a SWF file.
2. Caches the compiled SWF file on the server.
3. Returns the SWF file to the client.

Upon subsequent requests to the MXML file, the Flex server determines whether the MXML file has been modified since the previous request. If not, it returns the same SWF file from the cache. If the MXML file has been modified, the Flex server recompiles the file and returns an updated SWF file to the client.

The Flex programming model

Flex contains the Flex class library, and the MXML and ActionScript programming languages, as the following figure shows:



Also included in Flex, but not shown in this figure, are the Flex compilers and Flex debugger.

You write Flex applications using a combination of MXML and ActionScript. The MXML and ActionScript programing languages both give you the ability to access the Flex class library. Use MXML to declaratively define the application user interface elements and use ActionScript to define client logic and procedural control.

The Flex class library contains Flex components, managers, and behaviors. With the Flex component-based development model, developers can incorporate prebuilt components, create new components, or combine prebuilt components into composite components.

Relationship of the Flex class hierarchy to MXML and ActionScript

Flex is implemented as an ActionScript class library. That class library contains components (containers and controls), manager classes, data-service classes, and classes for all other features. You develop applications using the MXML and ActionScript languages with the class library.

MXML tags correspond to ActionScript classes or properties of classes. Flex parses MXML tags and compiles a SWF file that contains the corresponding ActionScript objects. For example, Flex provides the ActionScript Button class that defines the Flex Button control. In MXML, you create a Button control using the following MXML statement:

```
<mx:Button label="Submit"/>
```

When you declare a control using an MXML tag, you create an instance object of that class. This MXML statement creates a Button object, and initializes the `label` property of the Button object to the string "Submit".

An MXML tag that corresponds to an ActionScript class uses the same naming conventions as the ActionScript class. Class names begin with an uppercase letter, and uppercase letters separate the words in class names. Every MXML tag attribute corresponds to a property of the ActionScript object, a style applied to the object, or an event listener for the object. For a complete description of the Flex class library and MXML tag syntax, see the *Adobe Flex 2 Language Reference*.

Laying out your application

You declare a Flex user interface by using tags that represent user-interface components. There are two general types of components: controls and containers. Controls are form elements, such as buttons, text fields, and list boxes. Containers are rectangular regions of the screen that contain controls and other containers.

At the root of a Flex application is a single container, called the Application container, that represents the entire Flash Player drawing surface. This Application container holds all other containers, which can represent dialog boxes, panels, and forms.

A container has predefined rules to control the layout of its child containers and controls, including sizing and positioning. Flex defines layout rules to simplify the design and implementation of rich Internet applications, while also providing enough flexibility to let you create a diverse set of applications.

One advantage of having predefined layout rules is that your users will soon grow accustomed to them. That is, by standardizing the rules of user interaction, your users will not have to think about how to navigate the application, but can instead concentrate on the content that the application offers.

Another advantage is that you do not have to spend time defining navigation and layout rules as part of the design process. Instead, you can concentrate on the information that you want to deliver and the options that you want to provide for your users, and not worry about implementing all the details of user action and application response. In this way, Flex provides the structure that lets you quickly and easily develop an application with a rich set of features and interactions.

For more information on the Flex layout rules, see Chapter 8, "Sizing and Positioning Components," in *Flex 2 Developer's Guide*.

Using Flex Builder to lay out your application

You can use Flex Builder to build the user interface of a Flex application. You use Flex Builder to develop Flex and ActionScript applications by using visual coding and design tools. You can also write MXML, ActionScript, and CSS code in an editor that helps streamline your development work by providing code hinting and other code assistance features.

In the editor, you can switch into design mode and develop your applications visually, working with containers and components on the design canvas, using constraints to lay out your components, and view states to handle run-time user-interface transformations. You then build your projects by using a customizable compiler and debug the projects with integrated debugging tools.

For more information, see Chapter 5, "Building a Flex User Interface," in *Using Flex Builder 2*.

Defining a user interface in MXML

Your application can consist of one or more MXML files. Using multiple MXML files promotes code reuse, simplifies the process of building a complex application, and makes it easier for more than one developer to contribute to a project.

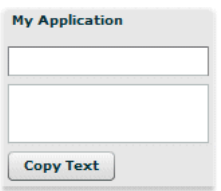
The following example is an MXML application that uses a Button control to trigger a copy of the text from a TextInput control to a TextArea control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- ?xml tag must start in line 1 column 1 -->

<!-- MXML root element tag. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Flex controls exist in a container. Define a Panel container. -->
    <mx:Panel title="My Application">

        <!-- TextInput control for user input. -->
        <mx:TextInput id="myInput" width="150" text=""/>

        <!-- Output TextArea control. -->
        <mx:TextArea id="myText" text="" width="150"/>

        <!-- Button control that triggers the copy. -->
        <mx:Button id="myButton" label="Copy Text"/>

    </mx:Panel>
</mx:Application>
```

The first line of this application specifies the XML declaration and must start in line 1, column 1 of the MXML file.

The second line begins with the `<mx:Application>` tag, the root element of a Flex application. This tag includes the Flex namespace declaration. The content between the beginning and end `<mx:Application>` tags defines the Flex application.

The following figure shows this application running in Flash Player:



As it is written, this example lays out the user interface, but does not yet contain the application logic to copy the input text from the TextInput control to the TextArea control. "Adding ActionScript to a Flex application" adds that logic.

Adding ActionScript to a Flex application

ActionScript follows the ECMA-262 Edition 4 (the specification written by the European Computer Manufacturers Association) unless otherwise noted. For more information on ActionScript, see Chapter 7, "Using ActionScript," on page 91.

You use ActionScript for the following purposes:

Handling events The Flex user interface is event-driven. For example, when a user selects a Button control, the Button generates an event. You handle events by defining functions in ActionScript called event listeners. Your event listener could open a window, play a SWF file, or perform whatever action is necessary for your application.

Handling errors You handle runtime errors in ActionScript. You can detect data validation errors and signal the error to the user, resubmit a request to the server, or perform some other actions based on your application.

Binding data objects to a Flex **control within an MXML statement** You can use data binding to populate a data model from a Flex component, populate a component from a data model, or copy data from one component to another.

Defining custom components You can derive custom components from the Flex component class hierarchy to create components specific to your application requirements.

The following example is a modification to the example in the previous section that adds an event listener for the `click` event of the Button control. An event listener is ActionScript code that is executed in response to a user action. The event listener in this example copies the text from the TextInput control to the TextArea control when the user selects the Button control:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- ?xml tag must start in line 1 column 1 -->

<!-- MXML root element tag. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Flex controls exist in a container. Define a Panel container. -->
    <mx:Panel title="My Application">

        <!-- TextInput control for user input. -->
        <mx:TextInput id="myInput" width="150" text=""/>

        <!-- Output TextArea control. -->
        <mx:TextArea id="myText" text="" width="150"/>

        <!-- Button control that triggers the copy. -->
        <mx:Button id="myButton" label="Copy Text"
            click="myText.text=myInput.text;"/>

    </mx:Panel>
</mx:Application>
```

The previous example inserts the ActionScript code directly into the MXML code. Although this technique works well for one or two lines of ActionScript code, for more complex logic you typically define your ActionScript in an `<mx:Script>` block, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- ?xml tag must start in line 1 column 1 -->

<!-- MXML root element tag -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[

            // Define an ActionScript function.
            private function duplicate():void {
                myText.text=myInput.text;
            }
        ]]>
    </mx:Script>

    <!-- Flex controls exist in a container. Define a Panel container. -->
    <mx:Panel title="My Application">

        <!-- TextInput control for user input. -->
        <mx:TextInput id="myInput" width="150" text=""/>

        <!-- Output TextArea control. -->
        <mx:TextArea id="myText" text="" width="150"/>

        <!-- Button control that triggers the copy. -->
        <mx:Button id="myButton" label="Copy Text"
            click="duplicate();"/>

    </mx:Panel>
</mx:Application>
```

In this example, you implement the event listener as an ActionScript function. Flex calls this function in response to the user selecting the Button control. This techniques lets you separate your MXML code from your ActionScript code. You can also choose to divide your application into multiple files to increase its modularity. For more information, see Chapter 7, “Using ActionScript,” on page 91.

Using data binding

Flex provides simple syntax for binding the properties of components to each other, or to a data model. In the following example, the value inside the curly braces ({ }) binds the `text` property of the TextArea control to the `text` property of a TextInput control. As the user enters text into the TextInput control, it is automatically copied to the TextArea control, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- ?xml tag must start in line 1 column 1 -->

<!-- MXML root element tag. -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- Flex controls exist in a container. Define a Panel container. -->
    <mx:Panel title="My Application">

        <!-- TextInput control for user input. -->
        <mx:TextInput id="myInput" width="150" text=""/>

        <!-- Output TextArea control. -->
        <mx:TextArea id="myText" text="{myInput.text}" width="150"/>

    </mx:Panel>
</mx:Application>
```

For more information, see Chapter 38, "Binding Data," in *Flex 2 Developer's Guide*.

Controlling application appearance

Flex defines a default appearance that you can use as is in an application, or modify to define your own specific appearance. As part of modifying the appearance, you can change some or all of the following:

Sizes Height and width of a component or application. All components have a default size. You can use the default size, specify your own size, or let Flex resize a component as part of laying out your application.

Styles Set of characteristics, such as font, font size, text alignment, and color. These are the same styles as defined and used with Cascading Style Sheets (CSS).

Skins Symbols that control a component's appearance. Skinning is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of images, SWF files, or class files that contain drawing API methods.

Behaviors Visible or audible changes to a Flex component that an application or user action triggers. Examples of behaviors are moving or resizing a component based on a mouse click.

View states In many Rich Internet Applications, the interface changes based on the task the user is performing. View states let you vary the contents and appearance of a component or application by modifying its base content.

Transitions Transitions define how a change of view state appears on the screen. You define a transition using the effect classes, in combination with several effects designed explicitly for handling transitions.

For more information on controlling appearance, see *Using Flex Builder 2* and *Flex 2 Developer's Guide*.

Working with data services

Flex is designed to interact with several types of services that provide access to local and remote server-side logic. For example, a Flex application can use AMF to connect to a web service that uses the Simple Object Access Protocol (SOAP), an HTTP URL that returns XML, or for Flex Data Services, a Java object that resides on the same application server as Flex. The MXML components that provide data access are called data service components. MXML includes the following types of data service components:

WebService Provides access to SOAP-based web services.

HTTPService Provides access to HTTP URLs that return data.

RemoteObject Provides access to Java objects (Java Beans, EJBs, POJOs) by using the AMF protocol. This option is only available with Flex Data Services or Macromedia ColdFusion MX 7.0.2.

How you choose to access data in your Flex application impacts performance. Because a Flex application is cached on the browser after the first request, data access is responsible for significantly affecting performance while the application runs. Flex provides several solutions for data delivery to the client. It delivers data through runtime services that invoke Java classes loaded in the Flex classpath, or sends proxy requests to web services or HTTP servers.

Using the WebService component enables you to use a SOAP-based approach, but it does not always yield the best performance. Also, the additional XML with the SOAP encoding requires more overhead than AMF does.

The performance of SOAP with web services is also dependent on your web services implementation. Different application servers use different web service back ends, so you might see performance differences depending on the implementation. The only way to understand how well your implementation will perform is to load-test your services.

Many times, the choice depends on your existing applications and how you choose to integrate them with your back end server-side resources. The performance of web services depends greatly on your application server's underlying implementation of the web services engine, so you should load-test to see how well it performs.

For more information, see Chapter 42, "Accessing Server-Side Data," in the *Flex 2 Developer's Guide*.

Separating your data model from your view

To cleanly separate the user interface, application-specific data, and data services, you can use Flex data models that store data between controls and data services. This type of three-tier design is applicable to both input data and data service results.

When you plan an application, you determine the kinds of data that the application must store and how that data must be manipulated. This helps you decide what types of data models you need. For example, suppose you decide that your application must store data about employees. A simple employee model might contain name, department, and e-mail address properties.

A Flex data model is an ActionScript object that contains properties that you use to store application-specific data. You can use a data model for data validation, and it can contain client-side business logic. You can define a data model in MXML or ActionScript. In the model-view-controller (MVC) design pattern, the data model represents the model tier.

You can define a data model in an MXML tag, ActionScript function, or an ActionScript class. A model written in MXML is useful for rapid development and simple data storage, but it does not provide any additional functionality, and you cannot set the data types of the model's properties. You should use an ActionScript-based class if you want to set data types and provide methods for additional functionality. In general, you should use MXML-based models for simple data structures and use ActionScript for more complex structures and client-side business logic.

For more information, see Chapter 37, "Representing Data," in the *Flex 2 Developer's Guide*.

About the Flex coding process

When you develop a Flex application, you use the same iterative process that you use for other types of web application files, such as HTML, JSP, ASP, and CFML.

Creating a useful Flex application is as easy as opening your favorite text editor, such as Flex Builder, typing some XML tags, compiling the file to a SWF file, deploying the SWF file, and requesting the SWF file's URL from a web browser.

Unlike a set of static HTML pages or HTML templates created using JSP, ASP, or CFML, the files in a Flex application are compiled into a single binary SWF file. Another major difference between a Flex application and a JSP, ASP, or ColdFusion application is that application logic resides in the client-side SWF file. JSP, ASP, and ColdFusion are templating systems in which application processing occurs on the server and data is dynamically added to an HTML template and delivered to the client in an HTML page. For more information, see "Moving to Flex from HTML" on page 68 and "Moving to Flex from an HTML templating environment" on page 69.

Because MXML files are ordinary XML files, you have a wide choice of development environments. You can develop in a simple text editor, a dedicated XML editor, or an integrated development environment (IDE) that supports text editing. Flex also provides tools for code debugging; for more information, see "Using Flex development tools" on page 70.

Moving to Flex from HTML

Although similar in some ways, developing a Flex application is significantly different from developing in HTML. HTML uses a page metaphor in which code is primarily written in a set of page files. What constitutes an application is really a set of separate HTML pages. Each page must be requested from the server and displayed individually. Assets such as images are loaded into the individual page that uses them when the page is requested. During development, you write code, save it, and display a page in a web browser.

The code for a Flex application is likely to be contained in more than one file to promote reusability, extensibility, and modularity. However, Flex compiles all files into a single SWF file.

The files that make up the application are compiled into the SWF file; however, the application can request data from external data sources at runtime. During development, you write code, save it, and display the entire application in a web browser.

Although Flex development is different from HTML development, you can easily incorporate a Flex application into an HTML page by using a wrapper. In the wrapper, you specify the name of a SWF file by using standard HTML `<object>` and `<embed>` tags. For more information, see Chapter 16, "Creating a Wrapper," in *Building and Deploying Flex 2 Applications*.

Moving to Flex from an HTML templating environment

In contrast to environments like JSP, ASP, and ColdFusion, Flex is not a templating system. MXML files are not templates that contain rules processed on the server to return a filled-in template. Instead, MXML code is compiled into a fully formed client application that is able to process server-side logic, and change what is displayed using its own client-side logic.

A Flex application does not go to the server every time the user interface must change or an event must be handled. Presentation logic and logic for server-side requests and responses is executed on the client as part of your application's SWF file. In Flex, changes to the user interface occur on the client, based on client-side code execution.

A Flex application makes HTTP requests to contact external data services, such as web services; this interaction with the server does not require you to refresh the application.

Moving to Flex from Flash Professional

Developing a Flex application is different from developing an application in Macromedia Flash Professional 8 from Adobe, even though in both environments the application is compiled into a SWF file. You create a Flex application in text files, which you can create and edit in a simple text editor or a more sophisticated development environment, such as Flex Builder. You compile your application into a SWF file, and then publish the application on a web or application server.

You create a Flash document file (a FLA binary file) in the Flash Authoring environment, and save it as a SWF file before publishing it to a website; it is usually referenced inside an HTML page. Flash Professional uses concepts such as the Timeline, animation frames, and layers for organizing and controlling an application's content over time. In Flex, you write the declarative parts of an application, such as user-interface components and connections to data sources, in MXML tags. You must use the tags in the proper hierarchy within the application, but there is no external notion of Timelines, frames, or layers. Just by using a few MXML tags, you can create a useful application with built-in behavior.

Although the development models for Flash and Flex are different, Flash is a powerful tool for creating custom components and visual assets that you can use in Flex applications. You can export files created in Flash in component package files called SWC files, which you can reference as custom tags in MXML files. You can also use the MXML `<mx:Image>` and `<mx:SWFLoader>` tags to import SWF files into a Flex application.

Using Flex development tools

Flex provides the following tools to help you test, debug, and tune your applications during the development process. The following table describes the Flex development tools:

Tool	Description
Flex Builder	Flex Builder is an integrated development environment (IDE) for developing applications using the Flex 2 SDK. The Flex Builder IDE provides tools that help you develop, design, and debug Flex applications, including an integrated incremental compiler and a step-through debugger. For more information, see *Using Flex Builder 2*.
Flash Debug Player	Flash Debug Player is a Flash Player that reports runtime errors, including errors for undeclared variables, uncaught runtime exceptions, and operating-system-specific networking errors. You can view errors in a trace window and send errors to a text file. For more information, see Chapter 11, "Running and Debugging Applications," in *Using Flex Builder 2*, or Chapter 12, "Using the Command-Line Debugger," in *Building and Deploying Flex 2 Applications*.
Command-line debugger	The Flex ActionScript debugger lets you open and debug ActionScript files that your Flex applications use. You can use the debugger from Flex Builder, or from a command line. For more information, see Chapter 11, "Running and Debugging Applications," in *Using Flex Builder 2*, or Chapter 12, "Using the Command-Line Debugger," in *Building and Deploying Flex 2 Applications*.
mxmlc	The Flex command-line compiler, mxmlc, is useful if you want to request SWF files in a web browser or in Flash Player. It is particularly useful for debugging SWF files with Flash Debug Player. For more information, see Chapter 11, "Running and Debugging Applications," in *Using Flex Builder 2*, or Chapter 9, "Using the Flex Compilers," in the *Flex 2 Developer's Guide*.
compc	The compc compiler generates a SWC file from MXML component source files or ActionScript component source files. For more information, see Chapter 9, "Using the Flex Compilers," in *Building and Deploying Flex 2 Applications*, and Chapter 10, "Building Projects," in *Using Flex Builder 2*.

CHAPTER 4

Building a Flex Data Services Application

4

When you use Flex Data Services to develop applications, you take advantage of its enhanced data services architecture. This topic contains an overview of Flex Data Services.

Contents

About Flex Data Services

You build on the functionality of Flex Software Development Kit (SDK) by adding Flex Data Services. Flex Data Services includes enterprise messaging support, and a greatly enhanced data services architecture. These features let you create and deploy enterprise-class applications that take full advantage of the rich presentation layer provided by Flex.

Flex Data Services executes on your Java application server or Java container to provide functionality in the following feature areas:

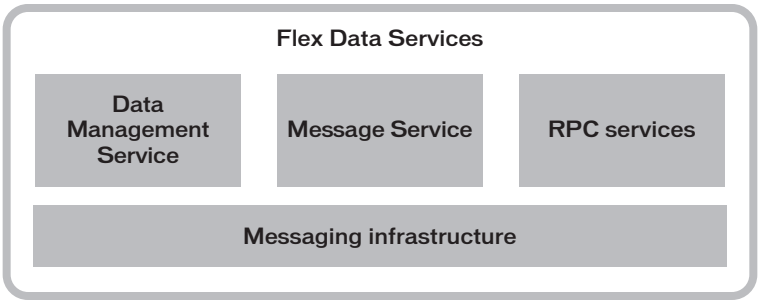
- Enhanced data services
  - Data sharing among multiple clients
  - Support for client-to-client data communication
  - Automated server data push
  - Authentication of client access to server resources
  - Data service logging
- Enhanced remote-procedure-call (RPC) functionality

This topic contains an overview of the features of Flex Data Services. For detailed information, see Chapter 42, "Accessing Server-Side Data," in *Flex 2 Developer's Guide*.

About Flex Data Management Service

The Flex Data Management Service is a Flex Data Services feature that lets you create applications that work with distributed data. This feature lets you build applications that provide data synchronization, data replication, and occasionally connected application services. Additionally, you can manage large collections of data and nested data relationships, such as one-to-one and one-to-many relationships, and use data adapters to integrate with data sources.

The Data Management Service, Message Service, and RPC services of Flex Data Services are based on an underlying messaging infrastructure, as the following example shows:



For more information on the Message Service, see "About the Flex Message Service" on page 73. For more information about RPC services, see "About RPC services" on page 74.

The messaging infrastructure lets a Flex application connect to a message destination, send messages to it, and receive messages from other messaging clients. Those messaging clients can be Flex applications or other types of clients, such as Java Message Service (JMS) clients. JMS clients can publish and subscribe to the same message destinations as Flex applications. This means that Flex applications can exchange messages with Java client applications.

The messaging infrastructure uses a server-side message service and a corresponding message API on Flex client-side applications. Flex client applications pass messages through a set of endpoints on the server, and the message service then routes the messages to a topic. The message service can then broadcast messages back to the endpoints and then to client applications that are subscribed to the topic.

Flex Data Services also expose the messaging infrastructure as the Flex Message Service so that you can build your own applications that take advantage of it. For more information, see "About the Flex Message Service" on page 73.

About the Flex Message Service

The Flex Message Service is based on established messaging standards and terminology. The Flex Message Service provides a client-side API and a corresponding server-side message service for creating Flex messaging applications. The Message Service also enables participation in Java Message Service (JMS) messaging.

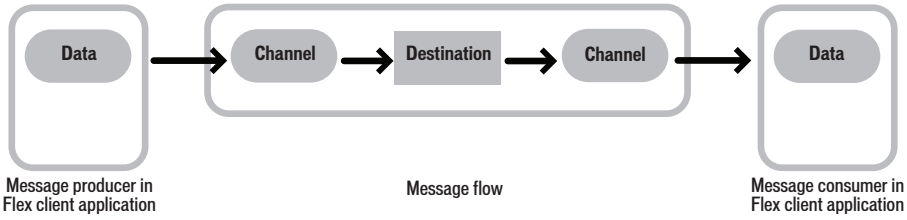
Messaging systems let separate applications communicate asynchronously as peers by passing packets of data called *messages* back and forth through a message service. A message is usually composed of a header and a body. The header contains an identifier and routing information. The body contains application data.

Applications that send messages are called *producers*. Applications that receive messages are called *consumers*. In most messaging systems, producers and consumers do not need to know anything about each other. Producers send messages to specific message destinations, and the message service routes the messages to the appropriate consumers.

A *message channel* connects producers and consumers to message destinations. To send messages over a particular channel, an application connects to the message endpoint associated with the message channel. A *message endpoint* is the code responsible for encoding data into messages, and decoding messages into a format that consumers can use. In some messaging systems, an endpoint can pass decoded messages to a message broker, which routes them to appropriate destinations.

A *message adapter* is code that acts as a conduit between the Flex Message Service and other messaging systems. For example, the Java Message Service (JMS) adapter is a message adapter that lets Flex applications subscribe to JMS topics and queues. This adapter lets a pure Java JMS message application share the same destination as a Flex application; Java applications can publish messages to Flex and Java code can respond to messages that Flex applications send.

The following example shows the flow of data from a message producer to a message consumer. Data is encoded as a message and sent over a channel to a destination. The message is then sent over a channel to the message consumer and decoded into data that the consumer can use.



Publish-subscribe messaging

Flex supports publish-subscribe messaging, also known as topic-based messaging. In Flex Message Service, a topic is synonymous with a message service destination. In publish-subscribe messaging, each message can have multiple consumers. You use this type of messaging when you want more than one consumer to receive the same message. Examples of applications that might use publish-subscribe messaging are auction sites, stock quote services, and other applications that require one message to be sent to many subscribers.

Producers publish messages to specific topics on a message server, and consumers subscribe to those topics to receive messages. Consumers can only consume messages that were published to a topic after they subscribed to the topic.

The following example shows a simple publish-subscribe message flow:

Using Flex Data Services in an application

Flex provides MXML and ActionScript APIs that let you use Flex Data Services in your applications. You can create applications that act as producers, consumers, or both. Flex Data Services applications send messages over channels declared on the Flex server to destinations also declared on the Flex server.

Flex Data Services uses XML configuration files to manage message topics and queues on the server. As part of configuring messaging, you can set the maximum message size. If a message exceeds this maximum value, multiple message batches are used, where the separate messages are reassembled by the client. This enables asynchronous data paging across the network.

About RPC services

Flex Data Services includes a Remoting Service and a Proxy Service for configuring RPC services. These features build on the RPC services available in the Flex 2 SDK.

Using RPC service components with the Flex 2 SDK only

You can use Flex 2 SDK without Flex Data Services to create applications that call HTTP services or web services directly without going through a server-side proxy service. You cannot use RemoteObject components without Flex Data Services. By default, Flash Player does not allow an application to receive data from a domain other than the domain from which the application was served. Therefore, an RPC service must either be in the same domain as the server hosting your application, or the remote server that hosts the RPC service must define a *crossdomain.xml* file.

A crossdomain.xml file is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from certain domains, or from all domains. The crossdomain.xml file must be in the web root of the server that the Flex application is contacting.

Using RPC service components with Flex Data Services

Use Flex Data Services when you want to provide enterprise functionality, such as proxying of service traffic from different domains, client authentication, whitelists of permitted RPC service URLs, server-side logging, localization support, and centralized management of RPC services. Flex Data Services also lets you use RemoteObject components to access remote Java objects without configuring them as SOAP-compliant web services.

When you use Flex Data Services, instead of contacting services directly, RPC service components contact destinations. *Destinations* are manageable service endpoints that you manage through a server-side XML-based configuration file.

About the development environment

You store Flex Data Services server-side code and application files in the directory structure of a standard web application on a J2EE-compliant application server or servlet container. The MXML deployment model is similar to that of JavaServer Pages (JSPs). You create an MXML file in the text editor of your choice, and place it in a web application directory that is accessible from a web browser. For example, you can place MXML files in the web application root directory or a subdirectory other than the reserved WEB-INF directory.

The directory structure of a typical Flex-enabled web application looks like the following:

Directory	Description
web_app (root directory or WAR root)	Contains the WEB-INF directory and all files that must be accessible by the user's web browser, such as MXML files, JSPs, HTML pages, Cascading Style Sheets, images, and JavaScript files. You can place these files directly in the web application root directory or in arbitrary subdirectories that do not use the reserved name WEB-INF.
/WEB-INF	Contains the standard web application deployment descriptor (web.xml) that configures Flex. This directory might also contain a web application deployment descriptor that is vendor specific.
/WEB-INF/flex	Contains Flex configuration files.
/WEB-INF/flex/user_classes	Contains custom ActionScript classes and MXML components.
/WEB-INF/lib	Contains Flex server code in Java Archive (JAR) files.
/WEB-INF/flex/frameworks	Contains the SWC component file, such as framework.swc and rpc.swc, that contains the Flex application framework files.

CHAPTER 5

Using Flex Charting Components

5

The ability to display data in a chart or graph can make data interpretation much easier for Flex application users. Rather than present a simple table of numeric data, you can display a bar, pie, line, or other type of chart using colors, captions, and a two-dimensional representation of your data.

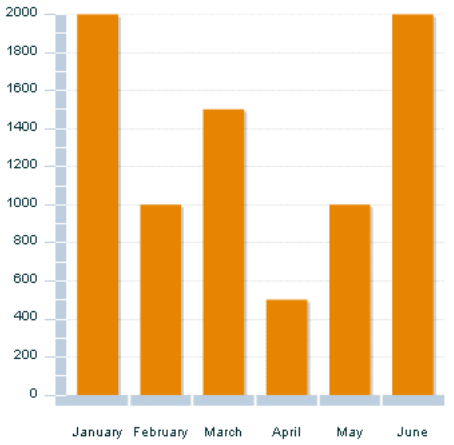
This topic introduces Flex charts, carting data, and the types of charts that you can create in Flex. For detailed information about creating and using charts, see Chapter 53, "Introduction to Charts," in *Flex 2 Developer's Guide*.

Contents

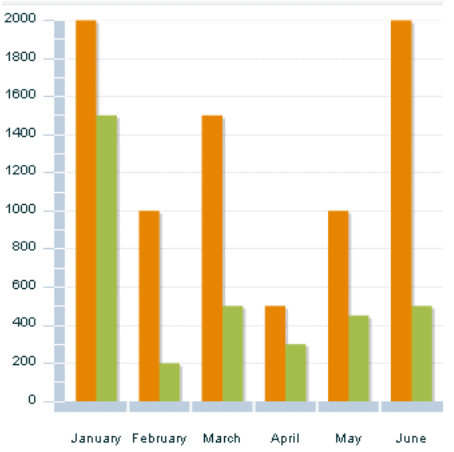
About charting

Data visualization lets you present data in a way that simplifies data interpretation and data relationships. Charting is one type of data visualization in which you create two-dimensional representations of your data. Flex supports some of the most common types of two-dimensional charts, such as bar, column, and pie charts, and provides you with a great deal of control over the appearance of the charts.

A simple chart shows a single data series, where a series is a group of related data points. For example, a data series might be monthly sales revenues, or daily occupancy rates for a hotel. The following chart shows a single data series that corresponds to sales revenue over six months:



Another chart might add a second data series. For example, you might include the profits over the same six months. The following chart shows two data series–one for sales and one for profit:



Defining chart data

The chart controls all take a `dataProvider` property that defines the data for the chart. A *data provider* is a collection of objects, similar to an array. The chart components use a flat, or list-based, data provider, similar to a one-dimensional array.

A data provider consists of two parts: a collection of data objects and an API. The data provider API is a set of methods and properties that a class must implement so that a Flex component recognizes it as a data provider.

The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple components from the same data provider, switch data providers for a component at runtime, and modify the data provider so that changes are reflected by all components that use the data provider.

For more information about using data providers in Flex, see Chapter 7, "Using Data Providers and Collections," in *Flex 2 Developer's Guide*.

Chart types

Flex supports many of the most common types of charts, including bar charts, line charts, pie charts, and others. This section describes the set of charts supplied with Flex. In addition to these chart types, you can also extend the CartesianChart control to create custom charts.

Area charts

You use the AreaChart control to represent data as an area bounded by a line connecting the values in the data. The area underneath the line is filled in with a color or pattern. You can use an icon or symbol to represent each data point along the line, or show a simple line without icons.

The following figure shows an example of an area chart:

Bar charts

You use the BarChart control to represent data as a series of horizontal bars whose length is determined by values in the data. You can use the BarChart control to represent a variety of chart variations, including clustered bars, overlaid stacked, 100% stacked, and high-low areas.

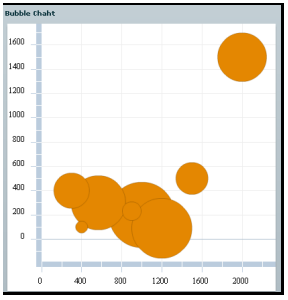
A bar chart is essentially a column chart rotated 90 degrees clockwise; therefore, bar charts and column charts share many of the same characteristics. For more information, see "Column charts" on page 81.

Bubble charts

You use the BubbleChart control to represent data with three values for each data point: a value that determines its position along the *x*-axis, a value that determines its position along the *y*-axis, and a value that determines the size of the chart symbol, relative to the other data points on the chart.

The `<mx:BubbleChart>` tag takes an additional property, `maxRadius`. This property specifies the maximum radius of the largest chart element, in pixels. The data point with the largest value is assigned this radius; all other data points are assigned a smaller radius based on their value relative to the largest value. The default value is `30` pixels.

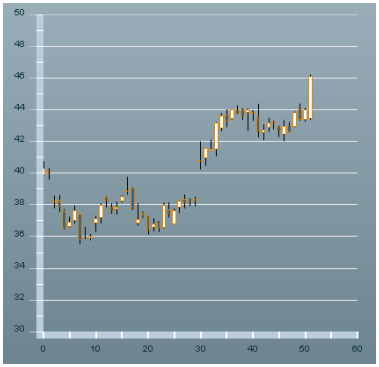
The following figure shows an example of a bubble chart:



Candlestick charts

The CandlestickChart control represents financial data as a series of candlesticks representing the high, low, opening, and closing values of a data series. The top and bottom of the vertical line in each candlestick represent the high and low values for the data point, while the top and bottom of the filled box represents the opening and closing values. Each candlestick is filled differently depending on whether the closing value for the data point is higher or lower than the opening value.

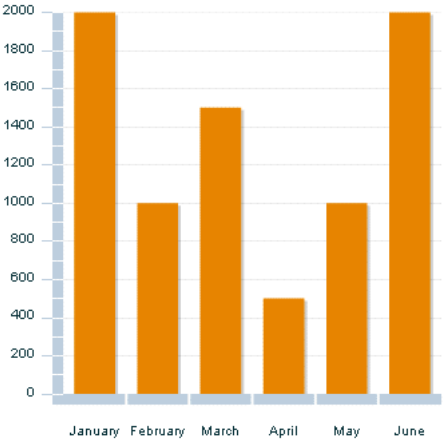
The following shows an example of a CandlestickChart chart:



Column charts

The ColumnChart control represents data as a series of vertical columns whose height is determined by values in the data. You can use the ColumnChart control to create several variations of column charts, including simple columns, clustered columns, overlaid stacked, 100% stacked, and high-low.

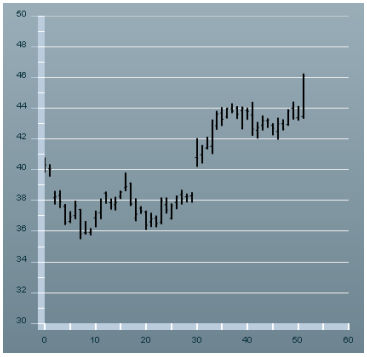
A simple chart shows a single data series, where a series is a group of related data points. For example, a data series might be monthly sales revenues, or daily occupancy rates for a hotel. The following chart shows a single data series that corresponds to the percentage growth in sales over four business quarters:



HighLowOpenClose charts

The HLOCChart control represents financial data as a series of lines representing the high, low, opening, and closing values of a data series. The top and bottom of the vertical line represent the high and low values for the data point, while the left tick mark represents the opening values and the right tick mark represents the closing values.

The HLOCChart control does not require a data point that represents the opening value. A related chart is the CandlestickChart control that represents similar data as candlesticks. You use the HLOCSeries with the HLOCChart control to define the data for HighLowOpenClose charts. The following example shows a HighLowOpenClose chart:



Line charts

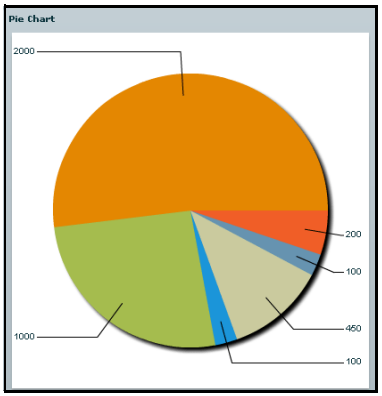
The LineChart control represents data as a series of points, in Cartesian coordinates, connected by a continuous line. You can use an icon or symbol to represent each data point along the line, or show a simple line without icons.

The following figure shows an example of a simple line chart:

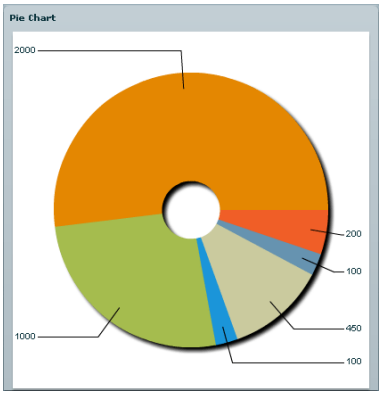
Pie charts

You use the PieChart control to define a standard pie chart. The data for the data provider determines the size of each wedge in the pie chart relative to the other wedges.

The following figure shows an example of a pie chart:



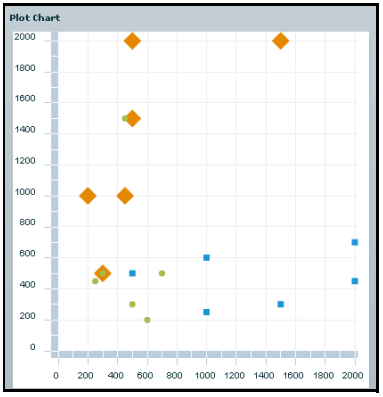
Flex lets you create doughnut charts out of PieChart controls. Doughnut charts are identical to pie charts, except that they have hollow centers and resemble wheels rather than filled circles. The following figure shows an example of a doughnut chart:



Plot charts

You use the PlotChart control to represent data in Cartesian coordinates where each data point has one value that determines its position along the *x*-axis, and one value that determines its position along the *y*-axis. You can define the shape that Flex displays at each data point with the data series' renderer.

The following figure shows an example of a plot chart with the CircleRenderer:



CHAPTER 6

Using MXML

6

MXML is an XML language that you use to lay out the user-interface for Adobe Flex applications. You also use MXML to declaratively define nonvisual aspects of an application, such as access to server-side data sources and data bindings between user-interface components and server-side data sources. This topic introduces MXML and describes how MXML relates to existing programming standards.

Contents

Using MXML

You use two languages to write Flex applications: MXML and ActionScript. MXML is an XML markup language that you use to lay out user-interface components. You also use MXML to declaratively define nonvisual aspects of an application, such as access to data sources on the server and data bindings between user-interface components and data sources on the server.

Like HTML, MXML provides tags that define user interfaces. MXML will seem very familiar if you have worked with HTML. However, MXML is more structured than HTML, and it provides a much richer tag set. For example, MXML includes tags for visual components such as data grids, trees, tab navigators, accordions, and menus, as well as nonvisual components that provide web service connections, data binding, and animation effects. You can also extend MXML with custom components that you reference as MXML tags.

One of the biggest differences between MXML and HTML is that MXML-defined applications are compiled into SWF files and rendered by Flash Player, providing a richer and more dynamic user interface than page-based HTML applications do.

You can write an MXML application in a single file or in multiple files. MXML also supports custom components written in MXML and ActionScript files.

Writing a simple application

Because MXML files are ordinary XML files, you have a wide choice of development environments. You can write MXML code in a simple text editor, a dedicated XML editor, or an integrated development environment (IDE) that supports text editing. Adobe supplies a dedicated IDE called Flex Builder that you can use to develop your applications.

The following example shows a simple "Hello World" application that contains just an `<mx:Application>` tag and two child tags, the `<mx:Panel>` tag and the `<mx:Label>` tag. The `<mx:Application>` tag is always the root tag of a Flex application. The `<mx:Panel>` tag defines a Panel container that includes a title bar, a title, a status message, a border, and a content area for its children. The `<mx:Label>` tag represents a Label control, a very simple user interface component that displays text.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" >

    <mx:Panel title="My Application" paddingTop="10" paddingBottom="10"
        paddingLeft="10" paddingRight="10" >

        <mx:Label text="Hello World!" fontWeight="bold" fontSize="24" />

    </mx:Panel>
</mx:Application>
```

Save this code to a file named hello.mxml. MXML filenames must end in a lowercase .mxml file extension. You can then compile and run the resultant SWF file.

The following figure shows the "Hello World" application rendered in a web browser window:

About XML encoding

The first line of the document specifies an optional declaration of the XML version. It is good practice to include encoding information that specifies how the MXML file is encoded. Many editors let you select from a range of file encoding options. On North American operating systems, ISO-8859-1 is the dominant encoding format, and most programs use that format by default. You can use the UTF-8 encoding format to ensure maximum platform compatibility. UTF-8 provides a unique number for every character in a file, and it is platform-, program-, and language-independent. If you specify an encoding format, it must match the file encoding you use. The following example shows an XML declaration tag that specifies the UTF-8 encoding format:

```
<?xml version="1.0" encoding="utf-8"?>
```

About the <mx:Application> tag

In addition to being the root tag of a Flex application, the `<mx:Application>` tag represents an Application container. A *container* is a user-interface component that contains other components and has built-in layout rules for positioning its child components. By default, an Application container lays out its children vertically from top to bottom. You can nest other types of containers inside an Application container, such as the Panel container shown above, to position user-interface components according to other rules. For more information, see Chapter 6, "Using Flex Visual Components," in *Flex 2 Developer's Guide*.

About MXML tag properties

The properties of an MXML tag, such as the `text`, `color`, and `fontSize` properties of the `<mx:Label>` tag, let you declaratively configure the initial state of the component. You can use ActionScript code in an `<mx:Script>` tag to change the state of a component at runtime. For more information, see Chapter 4, "Using ActionScript," in *Flex 2 Developer's Guide*.

The relationship of MXML tags to ActionScript classes

Adobe implemented Flex as an ActionScript class library. That class library contains components (containers and controls), manager classes, data-service classes, and classes for all other features. You develop applications using the MXML and ActionScript languages with the class library.

MXML tags correspond to ActionScript classes or properties of classes. Flex parses MXML tags and compiles a SWF file that contains the corresponding ActionScript objects. For example, Flex provides the ActionScript Button class that defines the Flex Button control. In MXML, you create a Button control using the following MXML statement:

```
<mx:Button label="Submit" />
```

When you declare a control using an MXML tag, you create an instance object of that class. This MXML statement creates a Button object, and initializes the label property of the Button object to the string Submit.

An MXML tag that corresponds to an ActionScript class uses the same naming conventions as the ActionScript class. Class names begin with an uppercase letter, and uppercase letters separate the words in class names. Every MXML tag attribute corresponds to a property of the ActionScript object, a style applied to the object, or an event listener for the object. For a complete description of the Flex class library and MXML tag syntax, see the *Adobe Flex 2 Language Reference*.

Using Flex Builder with MXML

Flex Builder includes a code editor that is the focal point for Adobe Flex application development in Flex Builder. As you work with MXML, the Flex Builder code editor helps to simplify and streamline your coding tasks by prompting you with hints that help you complete code expressions, checking for syntax errors, and formatting your code to improve readability.

For more information, see Chapter 9, "Code Editing in Flex Builder," in *Using Flex Builder 2*.

How MXML relates to standards

MXML uses standards extensively. This section describes how MXML relates to standards for the following:

- XML
- Event models
- Web services
- Java
- HTTP
- Graphics
- Cascading Style Sheets

XML standards

You write Flex applications in XML documents. XML documents use tags to define pieces of structured information and the relationships between them. In MXML, the `<mx:Application>` tag defines the root of an application. You define the parts of the application in child tags of the `<mx:Application>` tag. Examples of MXML tags include container tags, such as `<mx:VBox>`, which define rectangular regions of the user interface, and control tags, such as `<mx:TextInput>`, which define typical user interface controls.

Event model standards

The Flex event model is a subset of Document Object Model (DOM) Level 3 Events, a World Wide Web Consortium (W3C) working draft. DOM Level 3 defines an event system that allows platform- and language-neutral registration of event listeners, describes event flow through a tree structure, and provides basic contextual information for each event. MXML provides tag properties that let you specify ActionScript code to handle events. For example, the `click` event listener specified in the following example sends a web service request:

```
<mx:Button click="mywebservice.myOperation.send()”/>
```

Web services standards

Flex provides MXML tags for interacting with web services that define their interfaces in a Web Services Description Language (WSDL) document available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages.

Flex applications support web service requests and results that are formatted as Simple Object Access Protocol (SOAP) messages and are transported using the Hypertext Transfer Protocol (HTTP). SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

Java standards

Flex provides MXML tags for interacting with server-side Java objects, including plain old Java objects (POJOs), JavaBeans, and Enterprise JavaBeans (EJBs).

HTTP standards

Flex provides MXML tags for making standard HTTP GET and POST requests, and working with data returned in HTTP responses.

Graphics standards

Flex provides MXML tags for using JPEG, GIF, and PNG images in an application. Flex also provides tags for importing SWF files and Scalable Vector Graphics (SVG) files into an application.

Cascading Style Sheets standards

MXML styles are defined and used according to the W3C Cascading Style Sheets (CSS) standard. CSS provides a mechanism for declaring text styles and visual effects that you can apply to Flex components.

CHAPTER 7

Using ActionScript

7

Adobe Flex developers can use ActionScript to extend the functionality of their Flex applications. ActionScript provides flow control and object manipulation features that are not available in strict MXML. This chapter introduces ActionScript and explains how to use ActionScript in an MXML application.

For more information on ActionScript, see *Programming ActionScript 3.0*.

Contents

About ActionScript

ActionScript is the programming language for Adobe Flash Player. ActionScript includes built-in objects and functions, and lets you create your own objects and functions.

ActionScript is similar to the core JavaScript programming language. You don't need to know JavaScript to use ActionScript; however, if you know JavaScript, ActionScript will appear familiar to you. The following are some of the many resources available that provide more information about general programming concepts and the JavaScript language:

- The European Computers Manufacturers Association (ECMA) document ECMA-262 is derived from JavaScript and serves as the international standard for the JavaScript language.
- ActionScript is based on the ECMA-262 Edition 4 specification, which is available from www.ecma-international.org. ECMA-262 is also known as ECMAScript.

For more information about using ActionScript in Flex, see *Programming ActionScript 3.0*, which is included in the Flex documentation set.

Comparing ActionScript and ECMAScript (JavaScript)

ActionScript 3.0 features a full implementation of ECMAScript for XML (E4X), recently standardized as the ECMA-357 specification.

Using Flex Builder with ActionScript

Flex Builder includes a code editor that is the focal point for Adobe Flex application development in Flex Builder. As you work with ActionScript, the Flex Builder code editor helps to simplify and streamline your coding tasks by prompting you with hints that help you complete code expressions, checking for syntax errors, and formatting your code to improve readability.

For more information, see Chapter 9, "Code Editing in Flex Builder," in *Using Flex Builder 2*.

About ActionScript in MXML applications

ActionScript extends the capabilities of Flex application developers. With ActionScript, you can define event listeners, get or set component properties, handle callback functions, and create new classes, packages, and components.

You can use ActionScript in your Flex applications in the following ways:

- Insert ActionScript code blocks with the `<mx:Script>` tag. In these code blocks, you can add new functions, handle errors and events, and perform other tasks in your application or its supporting MXML files.
- Call global ActionScript functions that are stored in the system_classes directory structure.
- Reference external classes and packages in user_classes to handle more complex tasks. This lets you take advantage of ActionScript support for object-oriented programming concepts such as code reuse and inheritance.
- Use standard Flex components. The logic of components and helper classes in the Flex application model is based on ActionScript classes.
- Extend existing components with ActionScript classes.
- Create new components in ActionScript.
- Create new components in the Flash authoring environment (SWC files).

ActionScript compilation

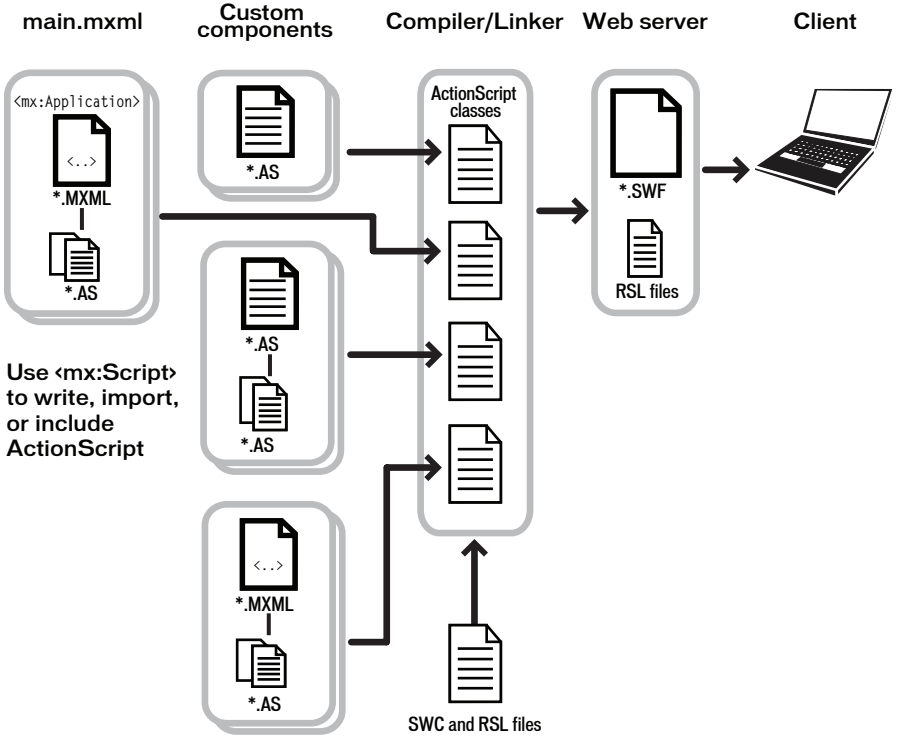
The logic of your Flex application can consist of ActionScript classes, MXML files, SWF files, and external components in the form of SWC components, MXML files, or ActionScript files. The end result of all the input files is a SWF file that is downloaded by Flash Player and played on the client's machine.

You can use ActionScript in code fragments within your MXML files. The Flex compiler transforms the main MXML file and its child files into a single ActionScript class. As a result, you cannot define classes or use statements outside of functions in MXML files and *included* ActionScript files.

You can reference *imported* ActionScript classes from your MXML application files, and those classes are added to the final SWF file.

When the transformation to an ActionScript file is complete, Flex links all the ActionScript components and includes those classes in the final SWF file.

The following example shows the source files used to generate a SWF file that your J2EE server sends to the client:



Using ActionScript in Flex applications

When you write a Flex application, you use MXML to lay out the user interface of your application, and you use ActionScript to handle the user interactions with the application. You can use a variety of methods to mix ActionScript and MXML. You can include inline ActionScript in MXML, or supplement a codeless MXML document with an external ActionScript controller class.

To use ActionScript in your Flex applications, you can add script blocks using the `<mx:Script>` tag or include external ActionScript files. In addition, you can import external class files or entire packages of class files for use by your MXML application. Flex extracts the ActionScript and creates a class file, which is linked to the final SWF file. For more information, see *Flex 2 Developer's Guide*.

NOTE	In general, Adobe recommends that you import ActionScript class files when possible rather than use `<mx:Script>` blocks in your MXML files or include snippets of ActionScript code from multiple files.

Using ActionScript blocks in MXML files

ActionScript blocks can contain ActionScript functions and variable declarations when used in MXML applications.

Statements and expressions are only allowed if they are wrapped in a function. In addition, you cannot define new classes or interfaces in `<mx:Script>` blocks. All ActionScript in the blocks is added to the enclosing file's class when Flex compiles the application.

When using an `<mx:Script>` block, you must wrap the contents in a CDATA construct. This prevents the compiler from interpreting the contents of the script block as XML, and allows the ActionScript to be properly generated. As a result, Adobe recommends that you write all your `<mx:Script>` open and close tags as the following example shows:

```
<mx:Script>
  <![CDATA[
    ...
  ]]>
</mx:Script>
```

The script within a given `<mx:Script>` tag is accessible from any component in the MXML file. The `<mx:Script>` tag must be located at the top level of the MXML file (within the Application tag or other top-level component tag). You can define multiple script blocks in your MXML files, but you should try to keep them in one location to improve readability.

The following example declares a variable and a function:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            public var z:Number;

            public function doSomething():void {
                z = z + 1;   // This must be in a function.
            }
        ]]>
    </mx:Script>

</mx:Application>
```

Using special characters in ActionScript

Special characters are any characters that might confuse the XML parser. The contents of `<mx:Script>` tags are treated as XML by Flex, but Flex does not parse text in a CDATA construct so that you can use XML-parsed characters such as angle brackets (< and >) and ampersand (&). For example, the following script that includes a less than (<) comparison in the for statement must be in a CDATA construct:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            public function changeText():void {
                for (var i:int=1; i<10; i++) {
                    ta1.text += "Hello world\n";
                }
            }
        ]]>
    </mx:Script>

    <mx:TextArea id="ta1" width="100" height="300"/>

    <mx:Button id="b1" label="Click Me" click="changeText()"/>

</mx:Application>
```

Referring to Flex components in ActionScript

When you refer to Flex components in ActionScript, the component must have an `id` property set. You then use the `id` to refer to that component.

For example, you can define a TextArea control in MXML using the `<mx:TextArea>` tag. To access that component's methods or properties, use dot syntax. The following example sets the value of the `text` property of the TextArea named *ta1*:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="setTextValue()">
    <mx:Script>
        <![CDATA[
            public function setTextValue():void {
                ta1.text = "Congratulations. You are a winner.";
            }
        ]]>
    </mx:Script>
    <mx:TextArea id="ta1"/>
</mx:Application>
```

You can refer to the current enclosing document or current object using the `this` keyword. For more information, see Chapter 4, "Using ActionScript," in *Flex 2 Developer's Guide*.

Including ActionScript code versus importing ActionScript classes

To make your MXML code more readable, you can also reference ActionScript files in your `<mx:Script>` tags, rather than insert large blocks of script. You can include or import ActionScript files.

There is a distinct difference between including and importing in ActionScript. *Including* is copying lines of code from one file into another. *Importing* is adding a reference to a class file or package so that you can access objects and properties defined by external classes. Files that you import must be found in the ActionScript classpath. Files that you include must be located relative to the application root or use an absolute path.

You use the `include` directive or the `<mx:Script source="filename">` tag to add ActionScript code snippets to your Flex applications.

You use `import` statements in an `<mx:Script>` block to add ActionScript classes and packages to your Flex applications.

For more information, see Chapter 4, "Using ActionScript," in *Flex 2 Developer's Guide*.

Creating ActionScript components

You create reusable components using ActionScript, and reference these components in your Flex applications as MXML tags. Components created in ActionScript can contain graphical elements, define custom business logic, or extend existing Flex components. They can inherit from any components available in Flex.

Defining your own components in ActionScript has several benefits. Components let you divide your applications into individual modules that you can develop and maintain separately. By implementing commonly used logic within custom components, you can build a suite of reusable components that you can share among multiple Flex applications.

In addition, you can base your custom components on the set of Flex components by extending from the Flex class hierarchy. You can create custom versions of Flex visual controls, as well as custom versions on nonvisual components, such as data validators, formatters, and effects.

For example, you can define a custom button, derived from the Flex Button control, as the following example shows:

```
// using_actionscript/myControls/MyButton.as
package myControls {
    import mx.controls.Button;
    public class MyButton extends Button {
        public function MyButton() {
        }
    }
}
```

In this example, you write your MyButton control to the MyButton.as file, and you store the file in the myControls subdirectory of the root directory of your Flex application. The package name of your component reflects its location. In this example, the component's package name is `myControls`.

You can reference your custom Button control from a Flex application file, such as MyApp.mxml, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
xmlns:cmp="myControls.*">

    <cmp:MyButton label="Click Me"/>

</mx:Application>
```

In this example, you first define the `cmp` namespace that defines the location of your custom component in the application's directory structure. You then reference the component as an MXML tag using the namespace prefix.

For more information, see *Creating and Extending Flex 2 Components*.

PART 3

Lessons

3

This part contains an a series of lessons designed to teach you the fundamentals of Flex. You can do all of the lessons, or only the ones that interest you.

CHAPTER 8

Create Your First Application

8

This lesson shows you how to compile and run a simple Flex application with Adobe Flex Builder. This lesson also introduces you to the concept of Flex Builder projects and shows you how to create projects. In Flex Builder, all Flex applications are contained within projects.

In this tutorial, you'll complete the following tasks:

Create the Lessons project

Before building a Flex application in Flex Builder, you must create a project. When you create a project in Flex Builder, a main MXML application file is created for you. You can add resources to a project, such as custom MXML component files, ActionScript files, and other assets that make up your Flex application.

1. Start Flex Builder and select File > New > Flex Project from the main menu.

   If you have the plug-in configuration of Flex Builder and you have a non-Flex-Builder perspective open in Eclipse, select New > Other > Flex > Flex Project.

   The New Flex Project wizard appears.

   The wizard guides you through the steps of creating a project.

2. In the opening screen, accept the Basic data option (the first option) and click Next.

   The next screen asks you to specify the name of the project and the location to store its files.

3. In the Project Name text box, enter **Lessons**.

   This is the name of your project. When you create a project, Flex Builder generates a main MXML application file based on the project name. Because a main application file uses the same name, you can't use spaces or special characters for the project name.

4. In the Project Contents area, make sure the Use Default Location option is selected.

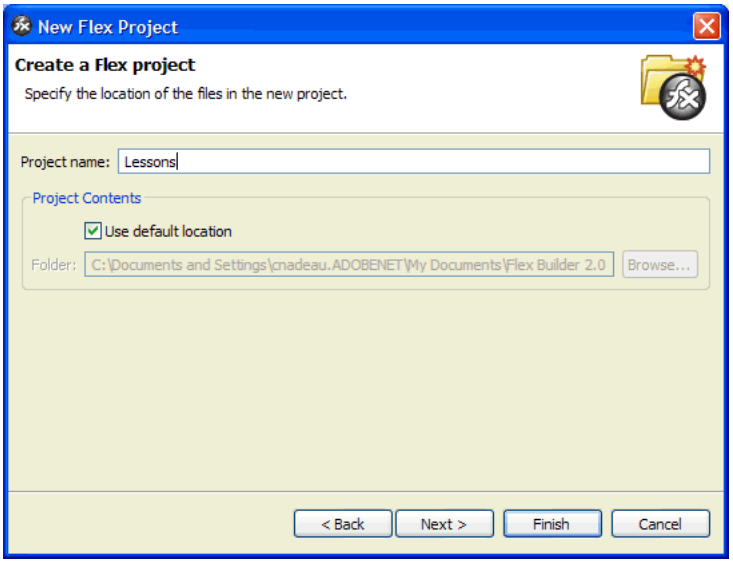
   The default location for your project files is in the following folder:

   Windows: C:\Documents and Settings*user_name*\My Documents\Flex Builder 2

   Macintosh: \Users*user_name*\Documents\Flex Builder 2

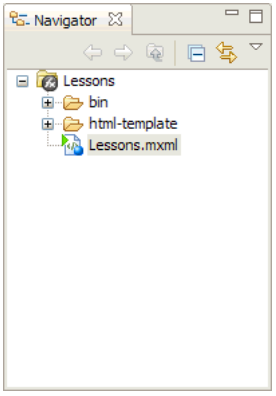
   Flex Builder will create the Lessons folder in that location for you.

   The following example shows the New Flex Project wizard:



5. Click Finish.

   Flex Builder creates a new project and displays it in the Navigator view.



The New Flex Project wizard automatically generates project configuration files, the output (bin) folder where your compiled SWF file will be placed, and the main application file, Lessons.mxml.

6. Ensure that the automatic build option is enabled in Flex Builder.

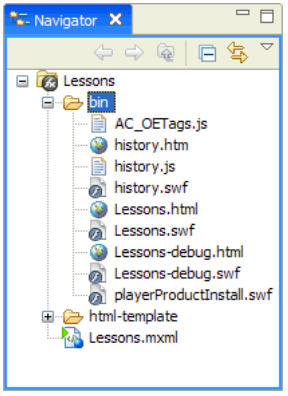
   This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Learn about compiling in Flex Builder

Before you compile and run an application in Flex Builder, it's helpful to review some key concepts.

By default, the standalone configuration of Flex Builder automatically compiles—or builds—the application when you add a file to the project or when you save a project file after editing it in Flex Builder. Automatic builds are disabled by default in the plug-in configuration of Flex Builder, but you can enable this option by selecting Project > Build Automatically.

After building an application, Flex Builder places the resulting Flash file (SWF) into the bin folder, which is the default folder in which compiled files are placed.



Flex Builder also generates an HTML wrapper file for the SWF file—in this case, Lessons.html—in case you want to run the SWF file in a web browser.

NOTE	The Flash Player 9 browser plug-in is required to run Flex 2 applications in a browser.

When you create a project, Flex Builder creates a main application file and names it based on the project name. An *application file* is an MXML file with an `<mx:Application>` parent tag. Your project can have several application files, but the main application file is the file Flex Builder compiles into the application SWF file during builds. You can designate another application file as the main application file to be built, but it's good practice to have only one application file per project.

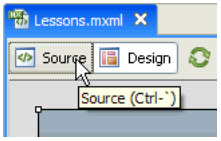
Now that you understand the basic concepts of compiling applications in Flex Builder, you can create a small application in Flex Builder, and then build and run it.

For more information about how projects are built, see "Understanding how projects are built" in *Using Flex Builder 2*.

Create and run an application

The steps in this section assume you created the Lessons project and that automatic builds are enabled. For more information, see "Create the Lessons project" on page 101.

1. If the Lessons.mxml file is not already open, double-click it in the Navigator view to open it in Flex Builder.
2. Switch to the MXML editor's Source mode by clicking the Source button in the document toolbar.



   Flex Builder inserted the following code in the Lessons.mxml file when it created it:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
</mx:Application>
```

3. Add a Panel container by entering the following `<mx:Panel>` tags between the opening and closing `<mx:Application>` tags:

```
<mx:Panel title="My Application" width="200" height="300">
</mx:Panel>
```

   Panel containers are the basic building blocks of many Flex layouts.

4. Add a Label control by entering the following `<mx:Label>` tag between the opening and closing `<mx:Panel>` tags:

```
<mx:Label text="Welcome to Flex!" mouseDownEffect="WipeRight"/>
```

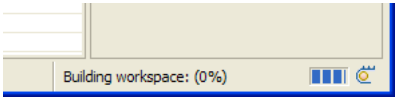
The final application should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel title="My Application" width="200" height="300">
        <mx:Label text="Welcome to Flex!" mouseDownEffect="WipeRight"/>
    </mx:Panel>
</mx:Application>
```

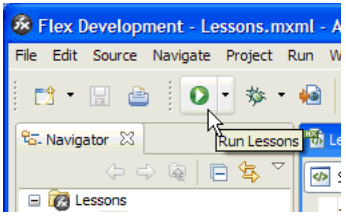
TIP	You can preview your layout by clicking the Design button in the document toolbar (see the image in step 2).

5. Save the file.

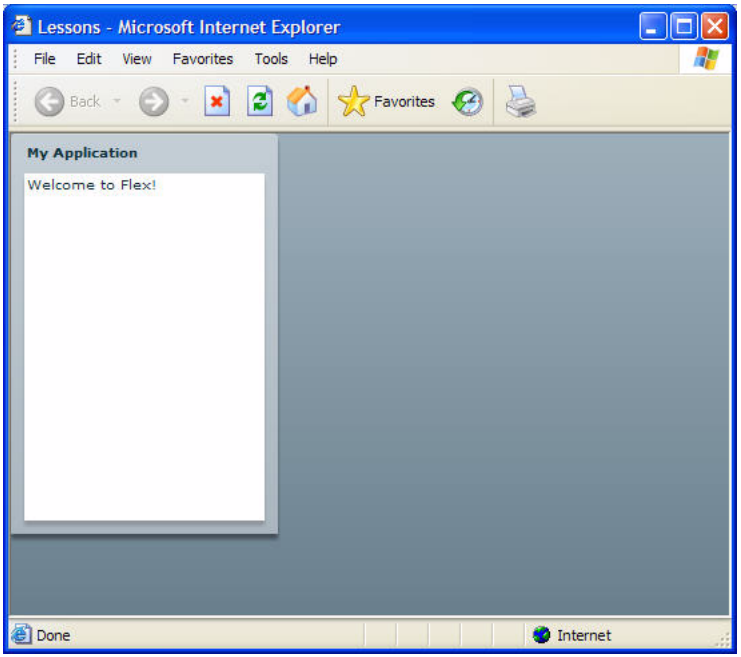
   Flex Builder automatically builds the application when you save a file. You can monitor the build progress with the indicator at the lower-right corner of the window. You can keep working while the application is building.



6. After the build is complete, click the Run button in the toolbar to start the application. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.



A browser opens and runs the application.



NOTE	The browser must have Flash Player 9 installed to run the application. You have the option of installing Flash Player 9 in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 9, in Flex Builder select Window › Preferences › General › Web Browser.

7. Click the "Welcome to Flex!" text to see the WipeRight effect.

   To learn more about effects, do the lesson in Chapter 13, "Use Behaviors," on page 151, or see Chapter 17, "Using Behaviors" in *Flex 2 Developer's Guide*.

You can deploy your application by uploading the SWF file generated by Flex Builder to a web server. You can also upload its HTML wrapper file (Lessons.html) to run the SWF file in a web browser. The files are located in the bin folder in your project folder.

In this lesson, you learned how to create a project in Flex Builder. You also learned how to compile and run a Flex application in Flex Builder. To learn more about this topic, see Chapter 10, "Building Projects" in *Using Flex Builder 2*.

CHAPTER 9

Retrieve and Display Data

9

To provide data to your application, Adobe Flex includes components designed specifically for interacting with HTTP servers, web services, or remote object services (Java objects). These components are called remote procedure call (RPC) service components.

Unlike web applications built with Adobe ColdFusion, PHP, or similar server technologies, Flex applications cannot connect directly to a database. They interact with data using services. For example, you can insert an HTTP service in a Flex file to interact with a ColdFusion file that retrieves data from a MySQL database, converts it to XML, and then feeds it to your Flex application.

In this lesson, you create a simple blog reader that retrieves recent posts and lets users read the first few lines of the posts. You use an RPC service component called HTTPService to retrieve data from an RSS feed, and then you bind the data to a Label, DataGrid, TextArea, and LinkButton control.

In this lesson, you'll complete the following tasks:

Set up your project

Before you begin this lesson, perform the following tasks:

- If you have not already done so, create the Lessons project in Adobe Flex Builder. For more information, see "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Review your access to remote data sources

For security reasons, applications running in Flash Player on a client computer can only access remote data sources if one of the following conditions is met:

- Your application's compiled SWF file is in the same domain as the remote data source.
- You use a proxy and your SWF file is on the same server as the proxy.

  Adobe Flex Data Services provides a complete proxy management system for Flex applications. You can also create a simple proxy service using a web scripting language such as ColdFusion, JSP, PHP, or ASP. For more information on creating your own proxy, see the following TechNote on the Adobe website at www.adobe.com/go/16520#proxy.
- A crossdomain.xml (cross-domain policy) file is installed on the web server hosting the data source.

  The crossdomain.xml file permits SWF files in other domains to access the data source. For more information on configuring crossdomain.xml files, see the following TechNote on the Adobe website at www.adobe.com/go/14213.

The data sources used in this lesson are located in a domain that has a crossdomain.xml setup. Therefore, Flash Player can access the remote data.

Insert and position the blog reader controls

In this section, you create the layout of your blog-reader application.

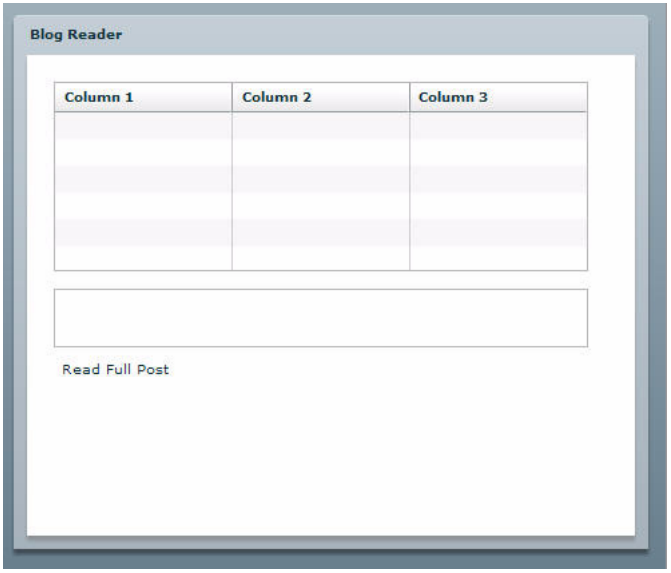
1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called BlogReader.mxml.

NOTE	For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the BlogReader.mxml file as the default file to be compiled by right-clicking (Control-click on Macintosh) the file in the Navigator view and selecting Set As Default Application from the context menu.
3. In MXML editor's Design mode, drag a Panel container into the layout from the Layout category of the Components view, and then set the following Panel properties in the Properties view:
   - Title: **Blog Reader**
   - Width: **475**
   - Height: **400**
   - X: **10**
   - Y: **10**
4. In the Design mode, add the following controls to the Panel container by dragging them from the Components view:
   - DataGrid
   - TextArea
   - LinkButton
5. Use the mouse to arrange the controls in the layout in a vertical, left-aligned column.
6. Select the DataGrid control and set the following properties:
   - Id: **dgPosts**
   - X: **20**
   - Y: **20**
   - Width: **400**
7. Select the TextArea control and set the following properties:
   - X: **20**
   - Y: **175**
   - Width: **400**
8. Select the LinkButton control and set the following properties:
   - Label: **Read Full Post**
   - X: **20**

- Y: 225

The layout should look like the following:



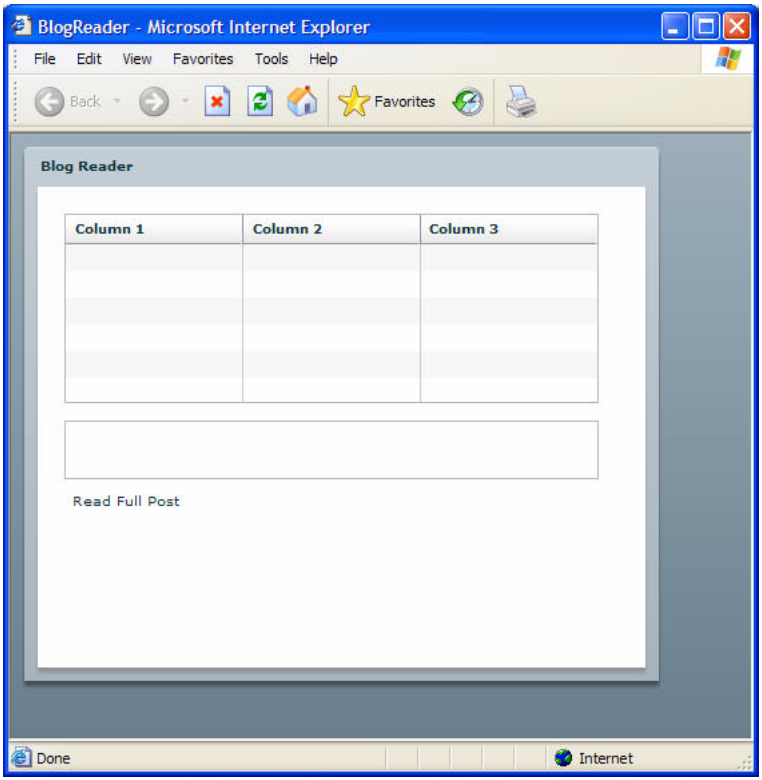
9. Switch to the editor's Source mode by clicking the Source button in the editor's toolbar.

The BlogReader.mxml file should contain the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel x="10" y="10" width="475" height="400" layout="absolute"
title="Blog Reader">
        <mx:DataGrid id="dgPosts" x="20" y="20" width="400">
            <mx:columns>
                <mx:DataGridColumn headerText="Column 1"
dataField="col1"/>
                <mx:DataGridColumn headerText="Column 2"
dataField="col2"/>
                <mx:DataGridColumn headerText="Column 3"
dataField="col3"/>
            </mx:columns>
        </mx:DataGrid>

        <mx:TextArea x="20" y="175" width="400"/>
        <mx:LinkButton x="20" y="225" label="Read Full Post"/>
    </mx:Panel>
</mx:Application>
```

10. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar to start the application. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.

    A browser opens and runs the application.



NOTE	The browser must have Flash Player 9 installed to run the application. You have the option of installing Flash Player 9 in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 9, in Flex Builder select Window › Preferences › General › Web Browser.

    The application doesn't display any blog information yet.

The next step is to retrieve information about recent blog posts. You can use an RPC service component called HTTPService to accomplish this task.

Insert a HTTPService component

For the blog reader in this lesson, you retrieve posts from Matt Chotin's blog at http://www.adobe.com/go/mchotinblog on a news aggregator website. Matt is a product manager on the Flex team and writes about Flex in his blog.

You can use the HTTPService component to access the blog's XML feed and retrieve information about recent posts. The component lets you send an HTTP GET or POST request, and then retrieve the data returned in the response.

1. In Source mode, enter the following `<mx:HTTPService>` tag immediately after the opening `<mx:Application>` tag:

```
<mx:HTTPService
  id="feedRequest"
  url="http://weblogs.macromedia.com/mchotin/index.xml"
  useProxy="false"/>
```

   The `url` property specifies the location of the requested file, in this case the RSS 2.0 feed of Matt Chotin's blog. As of this writing, the URL was still valid, but you should check to make sure it hasn't changed. You should use the latest RSS 2.0 URL listed on the right side of the blog at http://www.adobe.com/go/mchotinblog.

   NOTE If you want, you can also use the following alias in the component: http://www.adobe.com/go/flex_blogfeed.

   The `useProxy` property specifies that you don't want to use a proxy on a server. The domain where Matt's blog is located has a crossdomain.xml setup, so Flash Player can access the remote data sources on this server, including RSS feeds. For more information, see "Review your access to remote data sources" on page 108.

   The next step is to prompt the application to send a request to the specified URL. You decide to send the request automatically whenever the application starts, as follows.

2. In the `<mx:Application>` tag, add the following `creationComplete` property (in bold):

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute" creationComplete="feedRequest.send()">
```

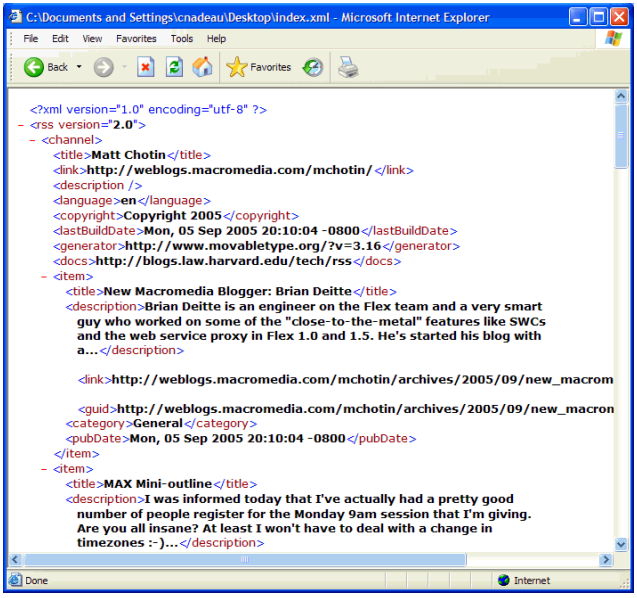
   When your application is finished starting up, the HTTPService component's `send()` method is called. The method makes an HTTP GET or POST request to the specified URL, and an HTTP response is returned. In this case, the RSS feed returns XML data.

   Next, you want to check if the application is retrieving the RSS feed successfully. You can do this by binding data to the Label control, as follows.

3. In the `<mx:Panel>` tag, replace the value of the `title` property ("Blog Reader") with the following binding expression (in bold):

```
title="{feedRequest.lastResult.rss.channel.title}"
```

This expression binds the title field to the Panel control. The expression reflects the structure of the XML. When XML is returned to a HTTPService component, the component parses it into an ActionScript object named lastResult. The structure of the lastResult object mirrors the structure of the XML document. To check the XML structure, download the RSS feed's XML file (at http://www.adobe.com/go/flex_blogfeed) and open it in Internet Explorer.



The general structure of the XML is as follows:

```
<rss>
  <channel>
    <title>
    other child nodes of <channel>
    <item>
      <title>
      other child nodes of <item>
    </item>
    ...
```

Some nodes have child nodes containing data, including the "title" child node of the channel node. The lastResult object of the HTTPService component (`feedRequest.lastResult`) reflects this structure:

```
feedRequest.lastResult.rss.channel.title
```

Your code should look like the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="feedRequest.send()" layout="absolute">

    <mx:HTTPService
        id="feedRequest"
        url="http://weblogs.macromedia.com/mchotin/index.xml"
        useProxy="false" />

    <mx:Panel x="10" y="10" width="475" height="400"
title="{feedRequest.lastResult.rss.channel.title}">

        <mx:DataGrid id="dgPosts" x="20" y="20" width="400">
            <mx:columns>
                <mx:DataGridColumn headerText="Column 1"
dataField="col1"/>
                <mx:DataGridColumn headerText="Column 2"
dataField="col2"/>
                <mx:DataGridColumn headerText="Column 3"
dataField="col3"/>
            </mx:columns>
        </mx:DataGrid>

        <mx:TextArea x="20" y="175" width="400"/>
        <mx:LinkButton x="20" y="225" label="Read Full Post"/>
    </mx:Panel>

</mx:Application>
```

4. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar to test the application.

   A browser opens and runs the application. The blog's title, Matt Chotin, should appear as the title of the Panel control, indicating that the application successfully retrieved data from the RSS feed.

NOTE	There may be a delay of a few seconds before the title appears while the application is contacting the server.

Populate a DataGrid control

In your application, you want the DataGrid control to display the titles of recent posts and the dates they were posted.

1. In Source mode, enter the following `dataProvider` property (in bold) in the `<mx:DataGrid>` tag:

   ```
   <mx:DataGrid x="20" y="20" id="dgPosts" width="400"
     dataProvider="{feedRequest.lastResult.rss.channel.item}">
   ```

   You want the XML node named item to provide data to the DataGrid control. This node is repeated in the XML, so it will be repeated in the DataGrid.

2. In the first `<mx:DataGridColumn>` tag, enter the following `headerText` and `dataField` property values (in bold):

   ```
   <mx:DataGridColumn headerText="Posts" dataField="title"/>
   ```

   You want the first column in the DataGrid control to display the titles of the posts. You do this by identifying the field in the XML that contains the title data, and then entering this field as the value of the `dataField` property. In the XML node specified in the `dataProvider` property (item), the child node called title contains the information you want.

3. In the second `<mx:DataGridColumn>` tag, enter the following `headerText`, `dataField` and `width` property values (in bold):

   ```
   <mx:DataGridColumn headerText="Date" dataField="pubDate" width="150"/>
   ```

   You want the second column in the DataGrid to display the dates of the posts. In this case, the field that contains the data is called `pubDate`.

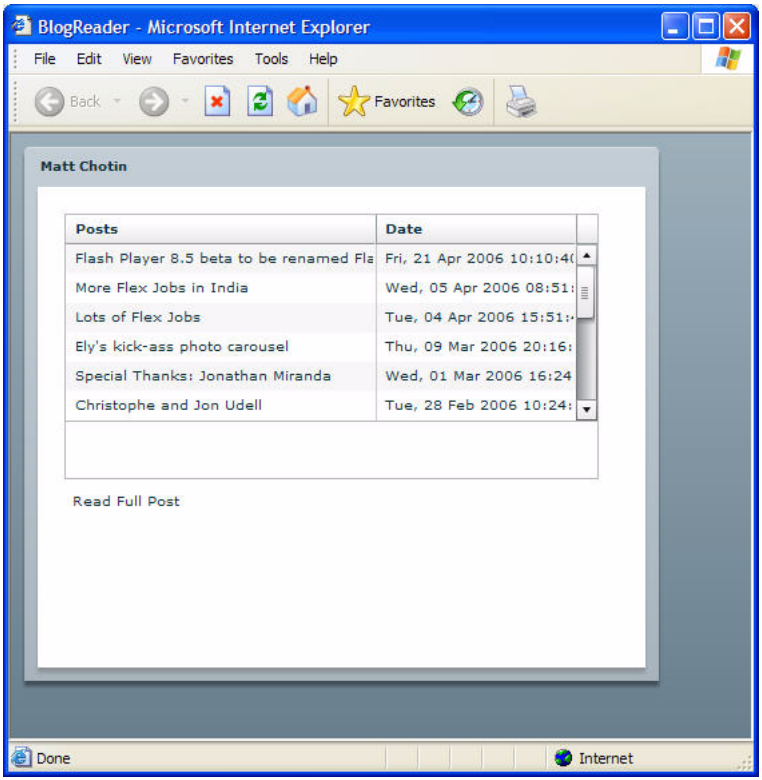
4. Delete the third `<mx:DataGridColumn>` tag because you don't need a third column.

The final application should look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="feedRequest.send()" layout="absolute">

    <mx:HTTPService
        id="feedRequest"
        url="http://weblogs.macromedia.com/mchotin/index.xml"
        useProxy="false"/>

    <mx:Panel x="10" y="10" width="475" height="400"
title="{feedRequest.lastResult.rss.channel.title}">

        <mx:DataGrid id="dgPosts" x="20" y="20" width="400"
dataProvider="{feedRequest.lastResult.rss.channel.item}">
            <mx:columns>
               <mx:DataGridColumn headerText="Posts" dataField="title"/>
               <mx:DataGridColumn headerText="Date" dataField="pubDate"
width="150"/>
            </mx:columns>
        </mx:DataGrid>

        <mx:TextArea x="20" y="175" width="400"/>
        <mx:LinkButton x="20" y="225" label="Read Full Post"/>
    </mx:Panel>

</mx:Application>
```

5. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button in the toolbar.

A browser opens and runs the application.



Blog titles and dates should appear in the DataGrid control, confirming that the application successfully retrieved data from the RSS feed and populated the control.

Display a selected item

When the user selects a post in the DataGrid control, you want the application to display the first few lines of the post in the TextArea control. In the item node of the XML feed, this information is contained in a field called description.

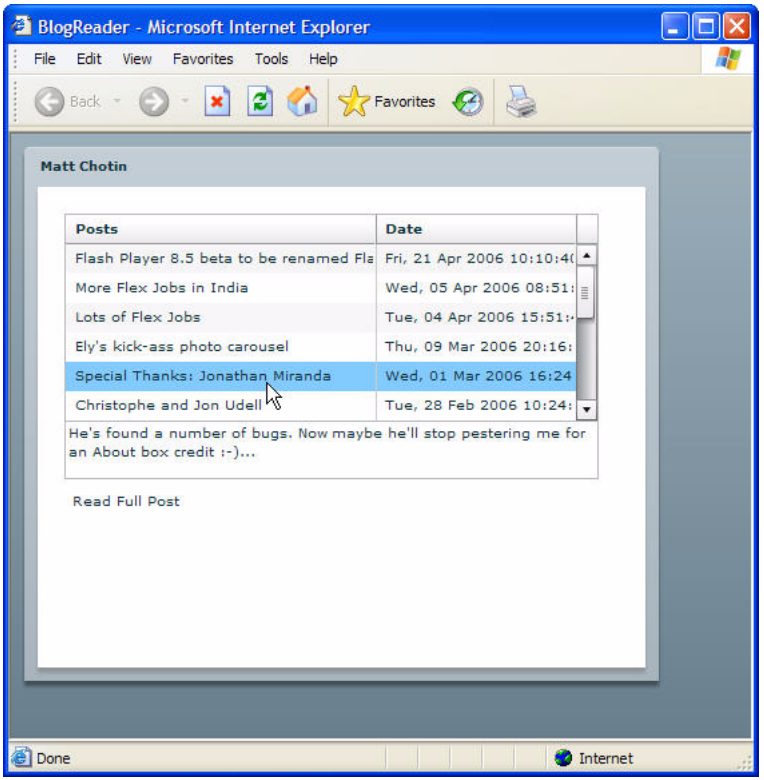
1. In Source mode, enter the following `htmlText` property (in bold) in the `<mx:TextArea>` tag:

```
<mx:TextArea x="20" y="175" width="400"
  htmlText="{dgPosts.selectedItem.description}" />
```

For each selected item in the DataGrid component (named dgPosts), the value of the description field is used for the value of the `htmlText` property. The `htmlText` property lets you display HTML formatted text.

2. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button in the toolbar.

   A browser opens and runs the application. Click items in the DataGrid control. The first few lines of each post should appear in the TextArea control.



Create a dynamic link

The RSS feed doesn't provide the full text of the posts, but you still want users to be able to read the posts if they're interested. While the RSS feed doesn't provide the information, it does provide the URLs to individual posts. In the item node of the XML feed, this information is contained in a field called link.

You decide to create a dynamic link that opens a browser and displays the full content of the post selected in the DataGrid.

1. In Source mode, change the `click` property in the `<mx:LinkButton>` tag so that the application source looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="feedRequest.send()" layout="absolute">

    <mx:HTTPService
        id="feedRequest"
        url="http://weblogs.macromedia.com/mchotin/index.xml"
        useProxy="false" />

    <mx:Panel x="10" y="10" width="475" height="400"
title="{feedRequest.lastResult.rss.channel.title}">

        <mx:DataGrid id="dgPosts" x="20" y="20" width="400"
dataProvider="{feedRequest.lastResult.rss.channel.item}">
            <mx:columns>
               <mx:DataGridColumn headerText="Posts" dataField="title"/>
               <mx:DataGridColumn headerText="Date" dataField="pubDate"
width="150" />
            </mx:columns>
        </mx:DataGrid>

        <mx:LinkButton x="20" y="225" label="Read Full Post"
click="navigateToURL(new URLRequest(dgPosts.selectedItem.link));"/>

        <mx:TextArea x="20" y="175" width="400"/>

    </mx:Panel>

</mx:Application>
```

The value of the link field of the selected item in the DataGrid control, `dgPosts.selectedItem.link`, is specified in the argument to the `navigateToURL()` method, which is called when the user clicks the LinkButton control. The `navigateToURL()` method loads a document from the specified URL in a new browser window.

NOTE The navigateToURL() method takes a URLRequest object as an argument, which in turn takes a URL string as an argument.

2. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button.

   A browser opens and runs the application. Click an item in the DataGrid control and then click the Read Full Post link. A new browser window should open and display the blog page with the full post.

In this lesson, you used an RPC service component called HTTPService to retrieve data from an RSS feed, and then you bound the data to a Label, DataGrid, TextArea, and LinkButton control. To learn more, see the following topics in *Flex 2 Developer's Guide*:

- Chapter 44, "Understanding RPC Components"
- Chapter 45, "Using RPC Components"

CHAPTER 10

Create a Constraint-based Layout

10

This lesson shows you how to create a constraint-based layout with Adobe Flex Builder. A constraint-based layout ensures that the components in your user interface adjust automatically when a user resizes the application window.

NOTE

You can achieve the same behavior using nested layout containers. For more information, see Chapter 15, "Using Layout Containers" in *Flex 2 Developer's Guide*.

In this tutorial, you'll complete the following tasks:

Set up your project

Before you begin this lesson, perform the following tasks:

- If not already done, create the Lessons project in Flex Builder. For more information, see "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Learn about constraint-based layouts in Flex

When a user resizes a Flex application window, you want the components in your layout to adjust automatically. A constraint-based layout adjusts the size and position of components when the user resizes the application window.

To create a constraint-based layout, you must use a container with a `layout` property set to absolute (`layout="absolute"`). This property gives you the flexibility of positioning and sizing components with absolute positioning while also providing you with the ability to set constraints that stretch and move the components when the container is resized.

NOTE

The Canvas container does not require the `layout="absolute"` property because its layout is absolute by default.

For example, if you want a TextInput text box to stretch when the user makes the application window wider, you can anchor the control to the left and right edges of the container so that the width of the text box is set by the width of the window.

In Flex, all constraints are set relative to the edges of the container. They cannot be set relative to other controls.

Now that you understand the basic concepts, you can create a simple layout and define layout constraints for it in Flex Builder.

Insert and position the components

The first step in creating a constraint-based layout is to position the components in a container with a `layout` property set to absolute. This property allows you to drag and position components anywhere in the container. For pixel-point accuracy, you can set *x* and *y* coordinates.

In this section, you insert and position the controls of a simple feedback form.

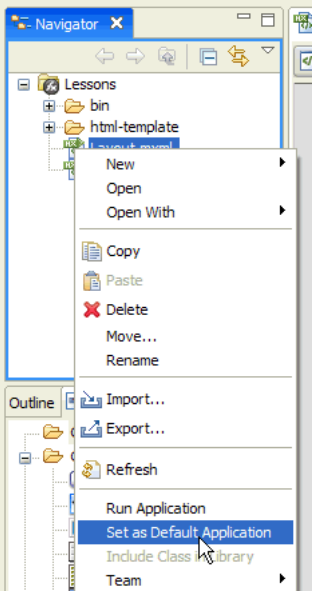
1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Layout.mxml.

   By default, Flex Builder includes the `layout="absolute"` property in the Application tag.

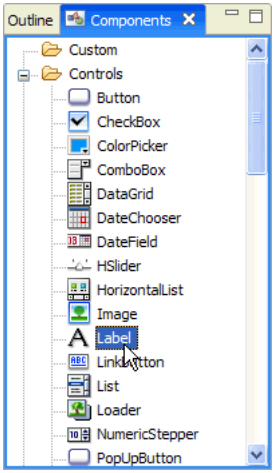
   NOTE

   For the purpose of these short tutorials, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the Layout.mxml file as the default file to be compiled by right-clicking (Control-click on Macintosh) the file in the Navigator view and selecting Set As Default Application from the context menu.



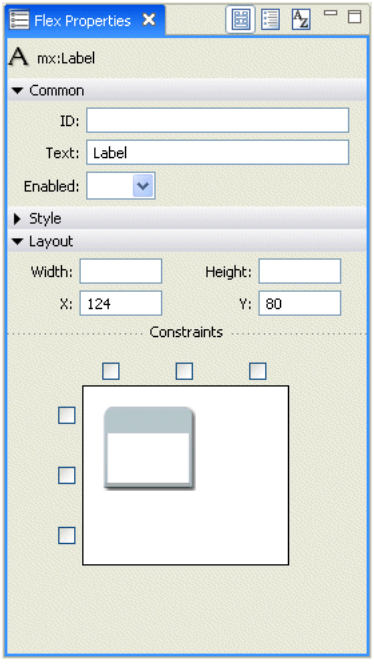
3. In the MXML editor's Design mode, add a Label and TextInput control to the Layout.mxml file by dragging them from the Components view (Window > Components).



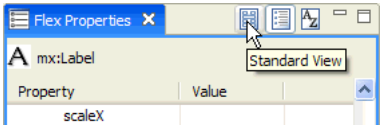
4. Use the pointer to position the Label and TextInput controls side-by-side about 60 pixels (two centimeters) from the top of the container.
5. In the Flex Properties view, expand the Common and the Layout categories of properties.

TIP	You may need to collapse the States view to see the Layout category.

Options for setting the common and layout properties appear.



If you see a table of properties instead of the previous view, click the Standard View button in the view's toolbar.



6. Select the Label control in the layout and set the following Label properties in the Flex Properties view:
   - Text: **Email**
   - X: **20**
   - Y: **60**
7. Select the TextInput control in the layout, and set the following TextInput properties:
   - X: **90**
   - Y: **60**
   - Width: **300**
8. Switch to the MXML editor's Source mode by clicking the Source button in the document toolbar.

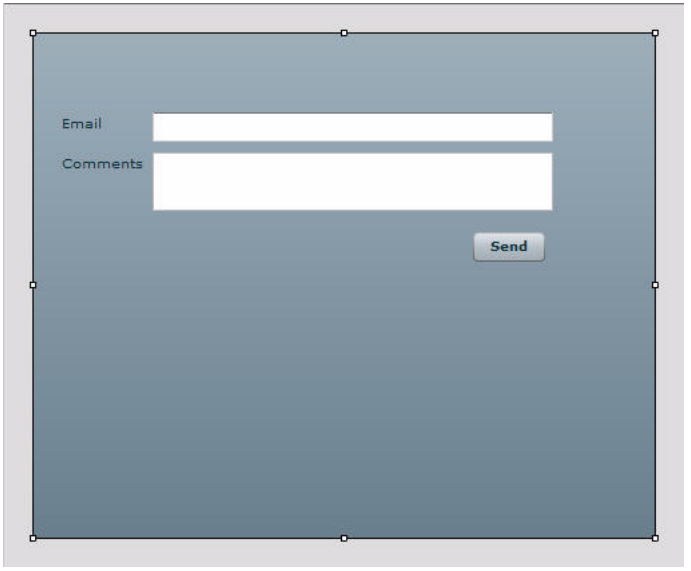
   The Layout.mxml file should contain the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Label x="20" y="60" text="Email"/>
    <mx:TextInput x="90" y="60" width="300"/>
</mx:Application>
```

9. Insert the remaining Flex controls by entering the `<mx:Label>`, `<mx:TextArea>`, and `<mx:Button>` tags after the `<mx:TextInput>` tag, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Label x="20" y="60" text="Email"/>
    <mx:TextInput x="90" y="60" width="300"/>
    <mx:Label x="20" y="90" text="Comments"/>
    <mx:TextArea x="90" y="90" width="300" />
    <mx:Button x="330" y="150" label="Send"/>
</mx:Application>
```

You can preview the layout by clicking the Design button in the toolbar. The layout should look similar to the following example:

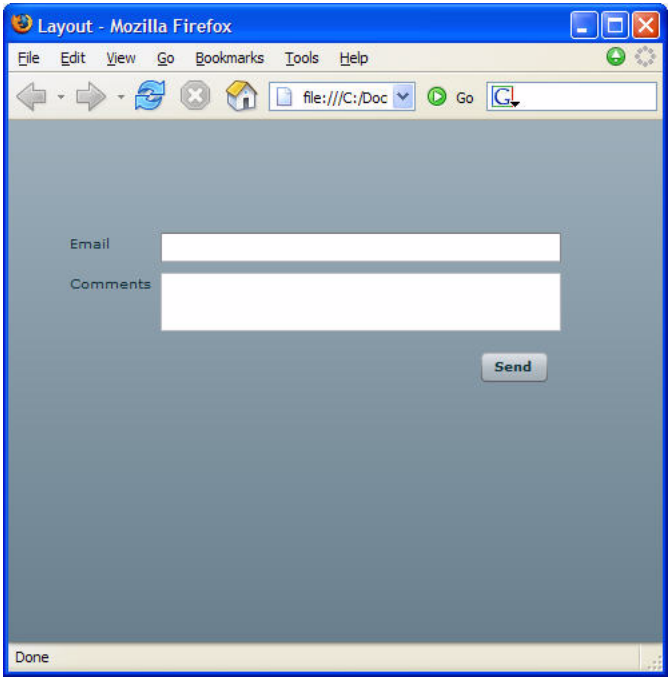


10. Save the file.

Flex Builder compiles the application.

11. Click the Run button in the toolbar. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.

    A browser opens and runs your small Flex application.

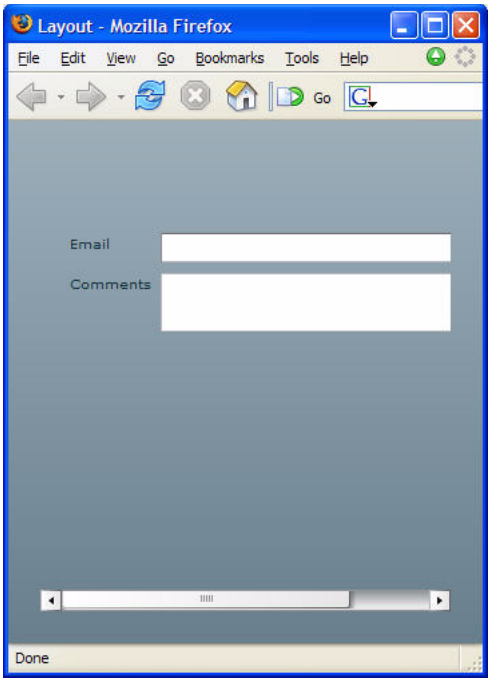


NOTE

The browser must have Flash Player 9 installed to run the application. You have the option of installing Flash Player 9 in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 9 in Flex Builder, select Window › Preferences › General › Web Browser.

12. Drag the edges of the browser window to make the application bigger and smaller.

    The components maintain their position relative to the left and top edges of the window, but they don't stretch or compress as you resize the browser window. For example, if you make the window too narrow, the Send button disappears, and the TextInput and TextArea controls are clipped.



NOTE

When content is clipped, Flex automatically provides users with a vertical or horizontal scrollbar to access the content.

The next step is to set layout constraints for the controls so that they adjust when the user resizes the application window.

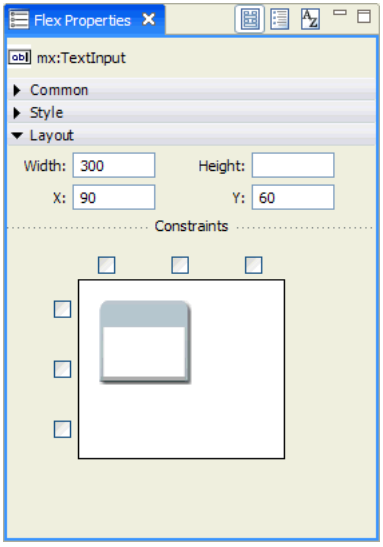
Define the layout constraints

After positioning the components in your layout, you define layout constraints so that the components adjust when a user resizes the application window.

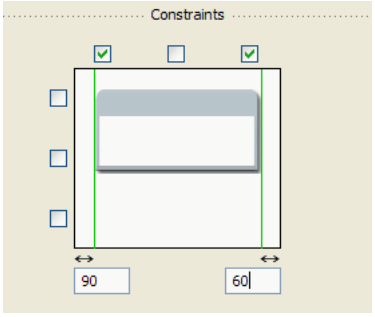
1. In the MXML editor's Design mode, select the TextInput control (the text box for the e-mail address).

2. In the Flex Properties view, ensure that the Layout category of properties is expanded.

   The Layout category contains options for setting anchors.



3. Define the layout constraints for the TextInput control by selecting the left and right anchor check boxes in the view, and then specifying **90** as the distance to maintain from the left window edge and **60** as the distance to maintain from the right edge, as follows:



   The two check boxes anchor the TextInput control to the left and right edges of the window. The numbers associated with the text boxes specify how far from the edges (in pixels) to anchor the controls.

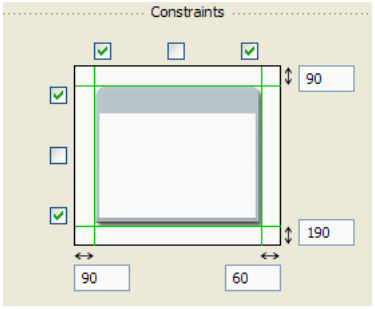
   The left anchor is necessary to fix the control in place so that it stretches or compresses when the user resizes the application horizontally. Without the left anchor to hold it in place, the control would slide to the left or right.

These constraints are expressed as follows in the MXML code:

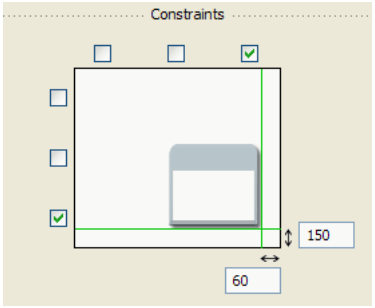
```
<mx:TextInput y="60" left="90" right="60"/>
```

4. In the editor's Design mode, select the TextArea control in the layout and, in the Flex Properties view, select the four corner check boxes and specify the following distances to maintain from the edges:
   - Left: **90**
   - Right: **60**
   - Top: **90**
   - Bottom: **190**

   The Layout category in the Flex Properties view for the TextArea control should look as follows:



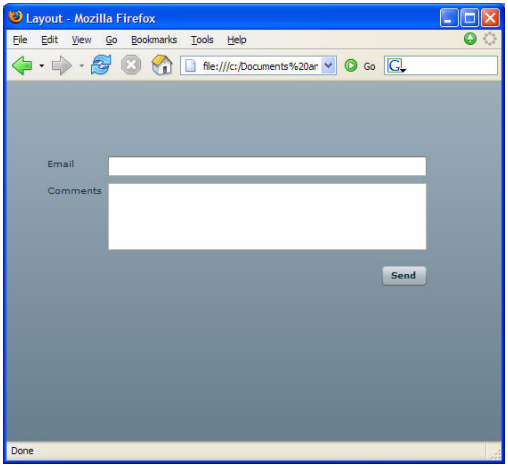
5. Select the Button control in the layout and, in the Flex Properties view, click the right and bottom anchor check boxes, and specify **60** as the distance to maintain from the right edge and **150** as the distance to maintain from the bottom edge, as follows:



The two check boxes anchor the Button control to the right and bottom edges. With no anchors to fix the control to the left and top edges, the control moves horizontally or vertically as the user resizes the application.

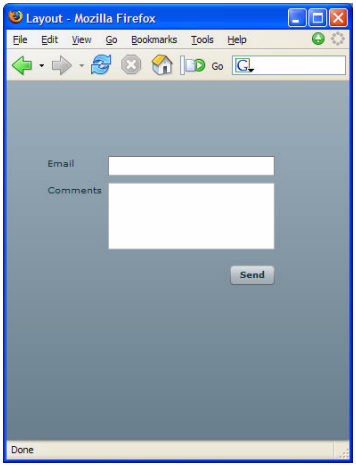
6. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar.

   A browser opens and runs your small Flex application.

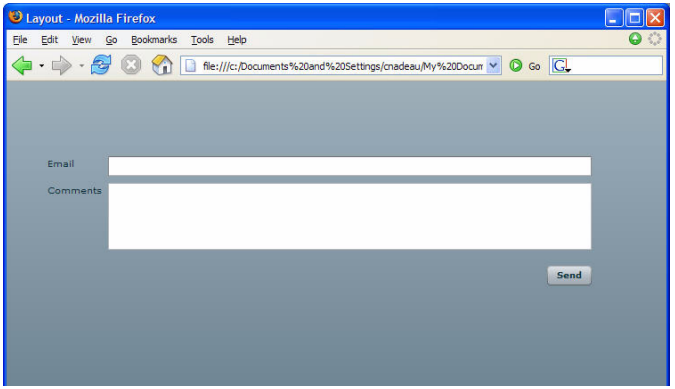


7. Drag the edges of the browser window to make the application bigger and smaller.

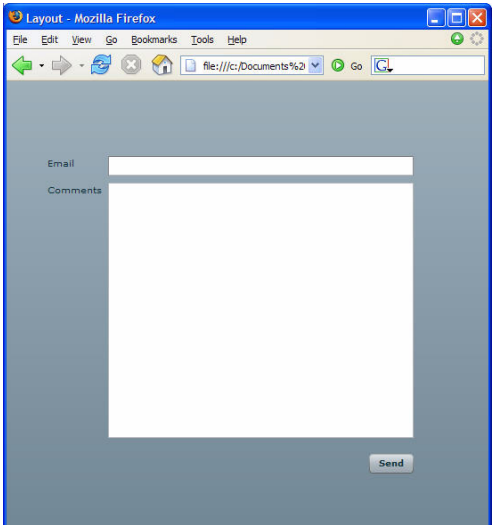
   For example, if you make the window narrower, the Send button moves to the left and the TextInput and TextArea text boxes become narrower.



If you make the browser window wider, the Send button moves to the right and the TextInput and TextArea text boxes become wider.



If you make the window taller, the Send button moves down and the TextArea becomes taller.



In this lesson, you learned how to create a constraint-based layout with Flex Builder. The following table summarizes the anchors to set for obtaining certain layout effects when the user resizes the application window.

Effect	Anchors
Maintain the component's position and size	None
Move the component horizontally	Right
Move the component vertically	Bottom
Move the component both horizontally and vertically	Right + Bottom
Resize the component horizontally	Left + Right
Resize the component vertically	Top + Bottom
Resize the component both horizontally and vertically	Left + Right + Top + Bottom
Center the component horizontally	Horizontal center
Center the component vertically	Vertical center

To learn more about this topic, see "Laying out your user interface" on page 133.

CHAPTER 11

Use List-based Form Controls

11

You can use list-based form controls such as a ComboBox, List, or HorizontalList in your Adobe Flex applications. After inserting this kind of control, you must populate it with items to display and values to submit for processing. In Flex, the controls are populated by data providers, which are collections of objects similar to arrays.

This lesson shows you how to populate list-based form controls with items to display and values to process.

In this lesson, you'll complete the following tasks:

Set up your project

Before you begin this lesson, perform the following tasks:

- If you haven't already done so, create the Lessons project in Adobe Flex Builder. See "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Insert and position controls

In this section, you create a simple layout containing a ComboBox control and a submit button.

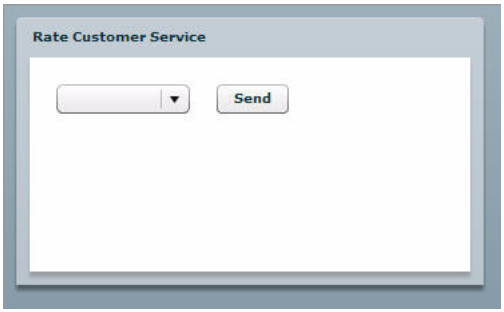
1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called ListControl.mxml.

NOTE	For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the ListControl.mxml file as the default file to be compiled by right-clicking (Control-click on Macintosh) the file in the Navigator view and selecting Set As Default Application from the context menu.
3. In the MXML editor's Design mode, drag a Panel container into the layout from the Layout category of the Components view, and then set the following Panel properties in the Properties view:
   - Title: **Rate Customer Service**
   - Width: **350**
   - Height: **200**
   - X: **10**
   - Y: **10**
4. Add the following controls to the Panel container by dragging them from the Controls category of the Components view (Window > Components):
   - ComboBox
   - Button
5. Select the ComboBox control in the layout and set the following ComboBox properties in the Flex Properties view:
   - ID: **cbxRating**
   - X: **20**
   - Y: **20**

   The ComboBox control doesn't list any items. You populate the list later.
6. Select the Button control and set the following Button properties in the Flex Properties view:
   - Label: **Send**
   - X: **140**
   - Y: **20**

The layout should look like the following example in Design mode:



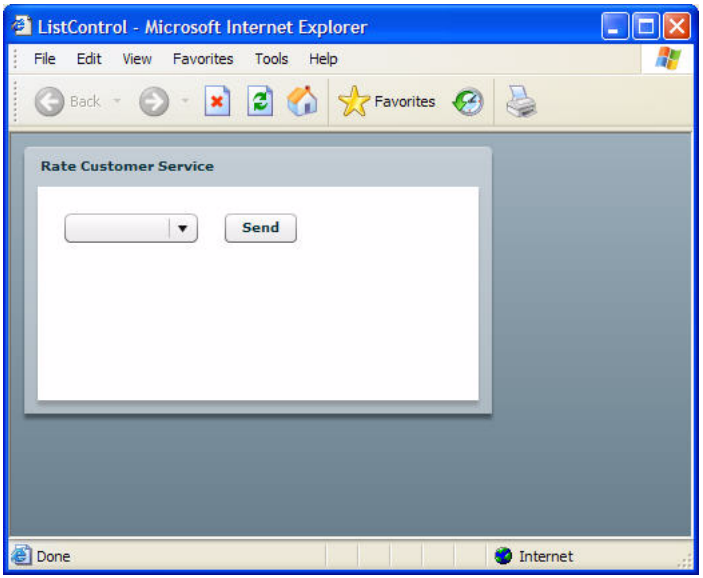
7. Switch to the editor's Source mode by clicking the Source button in the document toolbar.

   The ListControl.mxml file should contain the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel x="10" y="10" width="350" height="200" title="Rate Customer
Service">
        <mx:ComboBox id="cbxRating" x="20" y="20" width="100"></
mx:ComboBox>
        <mx:Button x="140" y="20" label="Send"/>
    </mx:Panel>
</mx:Application>
```

8. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.

A browser opens and runs your small Flex application.



NOTE	The browser must have Flash Player 9 installed to run the application. You have the option of installing Flash Player 9 in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 9, in Flex Builder select Window › Preferences › General › Web Browser.

9. Click the ComboBox control in the browser.

   The control doesn't list any items because you haven't defined its data provider yet.

Populate the list

You populate a list-based form control with the `<mx:dataProvider>` child tag. The `<mx:dataProvider>` tag lets you specify list items in several ways. The simplest method is to specify an array of strings, as follows.

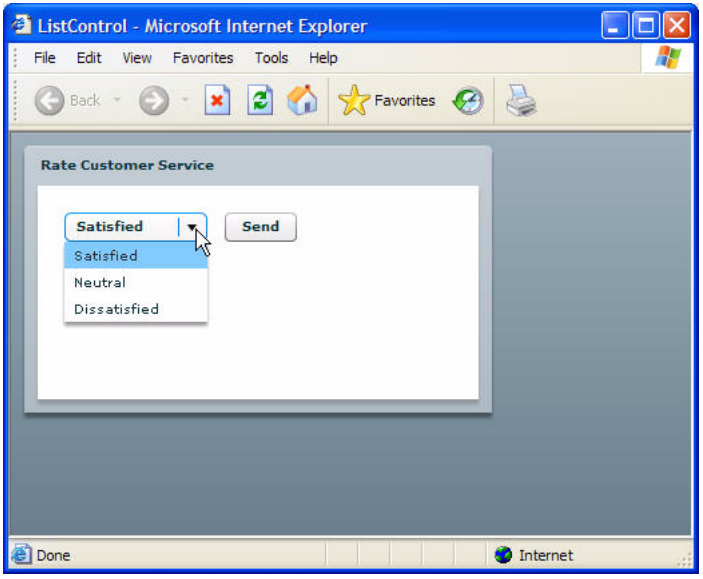
1. In the editor's Source mode, enter the `<mx:dataProvider>` tag between the opening and closing `<mx:ComboBox>` tag, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel x="10" y="10" width="350" height="200" title="Rate Customer
Service">
        <mx:ComboBox id="cbxRating" x="20" y="20" width="100">
            <mx:dataProvider>
                <mx:Array>
                    <mx:String>Satisfied</mx:String>
                    <mx:String>Neutral</mx:String>
                    <mx:String>Dissatisfied</mx:String>
                </mx:Array>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button x="140" y="20" label="Send"/>
    </mx:Panel>
</mx:Application>
```

2. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar.

   A browser opens and runs the application.

3. Click the ComboBox control to view the list of items.



If you want to access the value of the selected item in the ComboBox control, you can use the following expression in your code:

```
cbxRating.value
```

In this example, the `value` property of the ComboBox control (`cbxRating.value`) could contain Satisfied, Neutral, or Dissatisfied.

4. To test the control, insert the following tag after the `<mx:Button>` tag in the ListControl.mxml file:

```
<mx:Label x="20" y="120" text="{cbxRating.value}" />
```

The resulting application should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel x="10" y="10" width="350" height="200" title="Rate Customer
Service">
        <mx:ComboBox id="cbxRating" x="20" y="20" width="100">
            <mx:dataProvider>
                <mx:Array>
                    <mx:String>Satisfied</mx:String>
                    <mx:String>Neutral</mx:String>
                    <mx:String>Dissatisfied</mx:String>
                </mx:Array>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button x="140" y="20" label="Send"/>
        <mx:Label x="20" y="120" text="{cbxRating.value}"/>
    </mx:Panel>
</mx:Application>
```

The expression inside the curly braces ({ }) is a binding expression that copies the value of the ComboBox control's `value` property, `cbxRating.value`, into the Label control's `text` property. In other words, the `text` property of the Label control is specified by the value of the selected item in the ComboBox control.

5. Save the file, wait until Flex Builder finishes compiling, and run the application.

Select items in the ComboBox. The Label you inserted displays Satisfied, Neutral, or Dissatisfied depending on your selection.

Associate values with list items

You may want to associate values with list items in a form control in the same way you associate values with the SELECT form element in HTML. For example, to generate reports and statistics, you might want to associate the value of 5 with Satisfied, 3 with Neutral, and 1 with Dissatisfied.

To do this, you populate the ComboBox control with an array of Object components. The `<mx:Object>` tag lets you define a `label` property that contains the string to display in the ComboBox, and a `data` property that contains the data that you want to associate with the label.

1. In the editor's Source mode, replace the three `<mx:String>` tags with `<mx:Object>` tags, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel x="10" y="10" width="350" height="200" title="Rate Customer
Service">
        <mx:ComboBox id="cbxRating" x="20" y="20" width="100">
            <mx:dataProvider>
                <mx:Array>
                    <!-- These Object tags replace the String tags. -->
                    <mx:Object label="Satisfied" data="5"/>
                    <mx:Object label="Neutral" data="3"/>
                    <mx:Object label="Dissatisfied" data="1"/>
                </mx:Array>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button x="140" y="20" label="Send"/>
        <mx:Label x="20" y="120" text="{cbxRating.value}"/>
    </mx:Panel>
</mx:Application>
```

   If you want to access the value of the selected item in the ComboBox control, you can use the following expression in your code:

```
cbxRating.value
```

   The `value` property contains the value of the selected item. When a data field is specified, the `value` property refers to the data field, not the label field. In the example, the `cbxRating.value` property could contain the values 5, 3, or 1, depending on the user's selection.

2. Save the file, wait until Flex Builder finishes compiling, and then run the application.

   Select items in the ComboBox control in the browser. The testing Label control you inserted in the previous section displays the values 5, 3, or 1, depending on your selection.

To submit data for processing, you must use a remote procedure call (RPC) service component. For a tutorial, see Chapter 18, "Use Web Services," on page 205.

In this lesson, you inserted a list-based form control into your Flex application and populated it with data. To learn more, see Chapter 12, "Using Data-Driven Controls" in *Flex 2 Developer's Guide*.

CHAPTER 12

Use an Event Listener

12

When you develop Adobe Flex applications, event handling is one of the most basic and important tasks. Events let you know when something happens within a Flex application. They can be generated by user devices such as the mouse and keyboard, or other external input such as the return of a web service call. Events are also triggered when changes happen in the appearance or life cycle of a component, such as the creation or destruction of a component or when the component is resized.

You can respond to these events in your code by using *event listeners*. Event listeners are functions or class methods that you write to respond to specific events. They're also known as event handlers.

This lesson shows you how to use an event listener. You'll learn how to write one for a Button control, and then how to associate the listener with the Button's click event by using two different methods.

In this lesson, you'll complete the following tasks:

Set up your project

Before you begin this lesson, ensure that you perform the following tasks:

- If you have not already done so, create the Lessons project in Adobe Flex Builder. For more information, see "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Create a simple user interface

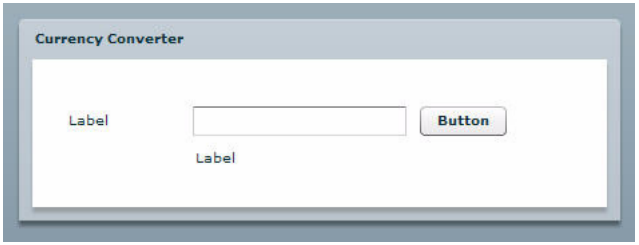
You decide to build a simple currency converter for your online store. You want the user to be able to specify a dollar amount and click a button to get the equivalent amount in yen. The first step is to design a simple user interface.

1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Events.mxml.

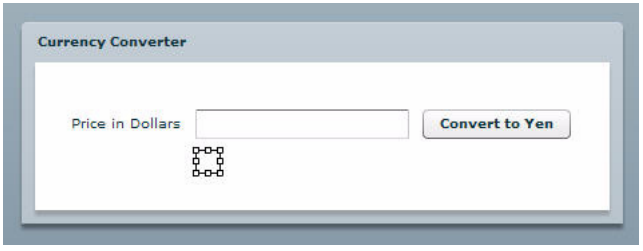
   NOTE: For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the Events.mxml file as the default file to be compiled by right-clicking (Control-click on Macintosh) the file in the Navigator view and selecting Set As Default Application from the context menu.
3. In the MXML editor's Design mode, drag a Panel container into the layout from the Layout category of the Components view, and then set the following Panel properties in the Properties view:
   - Title: **Currency Converter**
   - Width: **450**
   - Height: **150**
   - X: **20**
   - Y: **20**
4. Add two Label controls, a TextInput control, and a Button control to the Panel container by dragging them from the Components view (Window > Components).

5. Arrange the controls so that the layout is similar to the following example:



6. Select the first Label control and enter **Price in Dollars** as the value of its `text` property in the Flex Properties view.
7. Select the TextInput control and enter **txtPrice** as the value of its `id` property.
8. Select the Button control and set the following properties:
   - ID: **btnConvert**
   - Label: **Convert to Yen**
9. Select the second Label control (below the TextInput control) and do the following:
   - Clear the value of its Text property.
   - Enter **lblResults** as the value of its `id` property.
10. Fine-tune the position of the controls so that the layout looks as follows:



11. Switch to Source mode and examine the code generated by Flex Builder.

    Your code should look as follows (your coordinate values may vary):

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel x="20" y="20" width="450" height="150" layout="absolute"
        title="Currency Converter">
        <mx:Label x="25" y="37" text="Price in Dollars"/>
        <mx:Label x="120" y="65" id="lblResults"/>
        <mx:TextInput x="120" y="35" id="txtPrice"/>
       <mx:Button x="290" y="35" label="Convert to Yen" id="btnConvert"/
>
    </mx:Panel>
</mx:Application>
```

12. Save your file.

Write an event listener

Next, you write an event listener for the Convert to Yen button. You want the listener to consist of an ActionScript function that can calculate and display a specified dollar price in yen.

1. Switch to Source mode and place the insertion point immediately after the opening `<mx:Application>` tag.
2. Start typing **<mx:Script>** until the full tag is selected in the code hints, press Enter to insert the tag in your code, and then type the closing angle bracket (>) to complete the tag.

   Flex Builder enters an `<mx:Script>` script block that also includes a CDATA construct.

   NOTE When using an `<mx:Script>` script block, you should wrap the contents in a CDATA construct. This prevents the compiler from interpreting the contents of the script block as XML, and allows the ActionScript to be properly generated.

3. Enter or paste the following code in the CDATA construct:

```
public function convertCurrency():void {
  var rate:Number = 120;
  var price:Number = Number(txtPrice.text);
  if (isNaN(price)) {
    lblResults.text = "Please enter a valid price.";
  } else {
    price = price * rate;
    lblResults.text = "Price in Yen: " + String(price);
  }
}
```

The keyword `public` specifies the function's scope. In this case, the function is available throughout your code.

The keyword `void` specifies the function's return type. All ActionScript functions should specify a return type. The `convertCurrency` function returns no value.

The price entered by the user, `txtPrice.text`, is cast as a number and then validated to make sure the user entered a number. If the price is a number, the calculation is performed and the result is cast back to a string for display in the lblResults control.

In a real-world application, the value of the rate variable would be set at run time by calling a web service and retrieving the current exchange rate. As well, the result would be formatted as a currency.

4. Save the file.

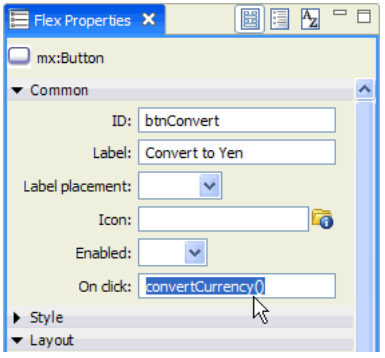
Associate the listener with an event with MXML

Associating a listener to an event, or *registering* it, means to include logic in your application that notifies the listener of a triggering event of a particular type from a particular source. For your application, you want the event listener to be notified of a click event from the Convert to Yen button. When notified that the event has occurred, the listener performs the currency calculation and displays the results.

One way to register the listener is to specify it as the value of the `click` property in the `<mx:Button>` tag.

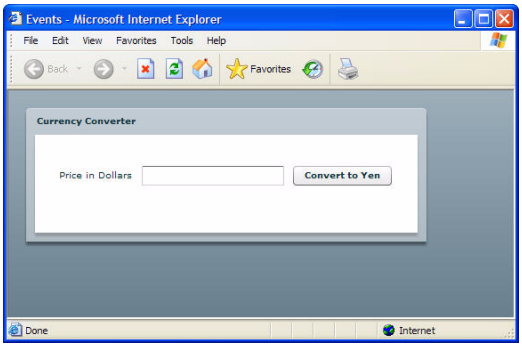
You can also use ActionScript to register the listener. For more information, see "Associate the listener with an event with ActionScript" on page 148.

1. In Design mode, select the Button control and enter **convertCurrency**() in the On Click text box in the Properties view.



2. Save the file, wait until Flex Builder compiles the application, and click Run in the toolbar. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.

   A browser opens and runs the application.



3. Enter a price and click the Convert to Yen button.

   The Label control below the TextInput control displays the price in yen.

Associate the listener with an event with ActionScript

You can use ActionScript to associate a listener with a specific event, such as a mouse click. When the event fires, the listener is notified and runs.

1. Switch to Source mode.
2. Delete the `click` property and its value in the `<mx:Button>` tag.
3. Declare an argument of type MouseEvent, or a subclass of the MouseEvent class, in the signature of your `convertCurrency` event listener as follows (in bold):

   ```
   public function convertCurrency(e:MouseEvent):void {
   ...
   ```

   In this example, the listener function takes an object of type flash.Events.MouseEvent, a subclass of the Event class. When a listener function is invoked, Flex implicitly creates a MouseEvent object for you and passes it to the listener function. Therefore, it is best practice to declare a MouseEvent object in the signature of your listener function. Accordingly, you declare an object of type MouseEvent, called e, in the signature of the `convertCurrency` function.

4. Enter the following function immediately before the `convertCurrency` function in the `<mx:Script>` tag:

```
public function createListener():void {
  btnConvert.addEventListener(MouseEvent.CLICK, convertCurrency);
}
```

When the user clicks the btnConvert button, the `convertCurrency` event listener is notified that a triggering event has occurred. The listener function performs the currency calculation and displays the results.

The script block should look as follows:

```
<mx:Script>
  <![CDATA[
  import flash.events.MouseEvent;

  public function createListener():void {
    btnConvert.addEventListener(MouseEvent.CLICK, convertCurrency);
  }

  public function convertCurrency(e:MouseEvent):void {
    var rate:Number = 120;
    var price:Number = Number(txtPrice.text);
    if (isNaN(price)) {
        lblResults.text = "Please enter a valid price.";
    } else {
      price = price * rate;
      lblResults.text = "Price in Yen: " + String(price);
    }
  }
  ]]>
</mx:Script>
```

5. In the `<mx:Application>` tag, enter the following property so that the `createListener()` function is called and the event listener is registered immediately after the application is created:

```
creationComplete="createListener();"
```

The final application should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute" creationComplete="createListener();">
    <mx:Script>
        <![CDATA[

        import flash.events.MouseEvent;

        public function createListener():void {
            btnConvert.addEventListener(MouseEvent.CLICK,
convertCurrency);
        }

        public function convertCurrency(e:MouseEvent):void {
            var rate:Number = 120;
            var price:Number = Number(txtPrice.text);
            if (isNaN(price)) {
                    lblResults.text = "Please enter a valid price.";
            } else {
                price = price * rate;
                lblResults.text = "Price in Yen: " + String(price);
            }
        }
        ]]>
    </mx:Script>

    <mx:Panel x="20" y="20" width="450" height="150" layout="absolute"
        title="Currency Converter">
        <mx:Label x="25" y="37" text="Price in Dollars"/>
        <mx:Label x="120" y="65" id="lblResults"/>
        <mx:TextInput x="120" y="35" id="txtPrice"/>
       <mx:Button x="290" y="35" label="Convert to Yen" id="btnConvert"/
>
    </mx:Panel>
</mx:Application>
```

6. Save the file, wait until Flex Builder compiles the application, and click Run in the toolbar.
7. Enter a price and click the Convert to Yen button.

   The Label control below the TextInput control displays the price in yen.

In this lesson, you wrote an event listener for a Button control, and then associated it with a button click event using two different methods. In the first method, you associated it by specifying it in the `click` property in the Button control's MXML tag. In the second method, you associated the listener by writing an ActionScript function. To learn more, see Chapter 5, "Using Events" in *Flex 2 Developer's Guide*.

CHAPTER 13

Use Behaviors

13

Adobe Flex behaviors let you add animation and motion to your application in response to user or programmatic action. A behavior is a combination of a trigger paired with an effect. A trigger is an action, such as a mouse click on a component, a component getting focus, or a component becoming visible. An effect is a visible or audible change to the target component that occurs over a period of time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component.

This lesson shows you how to add behaviors to a Flex user interface. It shows you how to use MXML to create behaviors, how to invoke an effect from a different component, and how to combine more than one effect to create a composite effect.

In this lesson, you'll complete the following tasks:

Set up your project

Before you begin this lesson, ensure that you perform the following tasks:

- If you have not already done so, create the Lessons project in Adobe Flex Builder. For more information, see "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Create a behavior

You decide to create a behavior so that a button glows when a user clicks it. You want the glow to be green, last one and a half seconds, and leave the button a pale green to indicate it was clicked.

1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Behaviors.mxml.

   NOTE

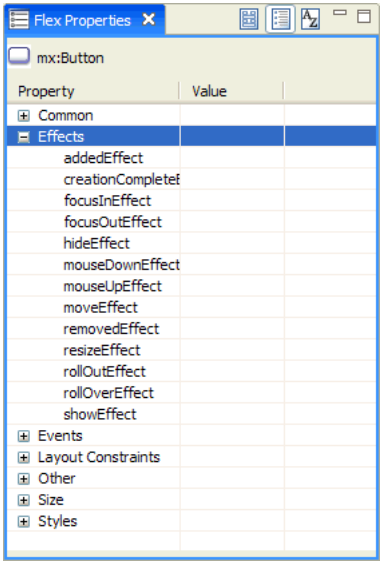
   For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the Behaviors.mxml file as the default file to be compiled by right-clicking (Control-click on Macintosh) the file in the Navigator view and selecting Set As Default Application from the context menu.
3. In the MXML editor's Source mode, define a Glow effect by entering the following tag after the opening `<mx:Application>` tag:

   ```
   <mx:Glow id="buttonGlow" color="0x99FF66" alphaFrom="1.0" alphaTo="0.3"
     duration="1500"/>
   ```

   The Glow effect starts fully opaque and gradually becomes more transparent, but not fully transparent. A pale glow persists after the effect has played to indicate that the button was clicked.
4. In Design mode, drag a Panel container into the layout from the Layout category of the Components view, and then set the following Panel properties in the Properties view:
   - Width: **200**
   - Height: **300**
   - X: **10**
   - Y: **10**
5. Drag a Button control into the Panel from the Controls category of the Components view, and then set the following Button properties in the Properties view:
   - ID: **myButton**
   - Label: **View**
   - X: **40**
   - Y: **60**

6. In the Properties view, click the Category View button in the toolbar to view the properties as a table, and then locate the Effects category of properties.



This category lists the triggers for the Button control.

7. Assign your Glow effect to the `mouseUpEffect` trigger by entering the effect's ID in curly braces as the value of the trigger, as follows:

   - mouseUpEffect: {**buttonGlow**}

   The curly braces ({ }) are necessary because effects are assigned to their triggers using data binding.

   In Source mode, the `<mx:Button>` tag should look as follows:

```
<mx:Button x="40" y="60" label="View" id="myButton"
  mouseUpEffect="{buttonGlow}"/>
```

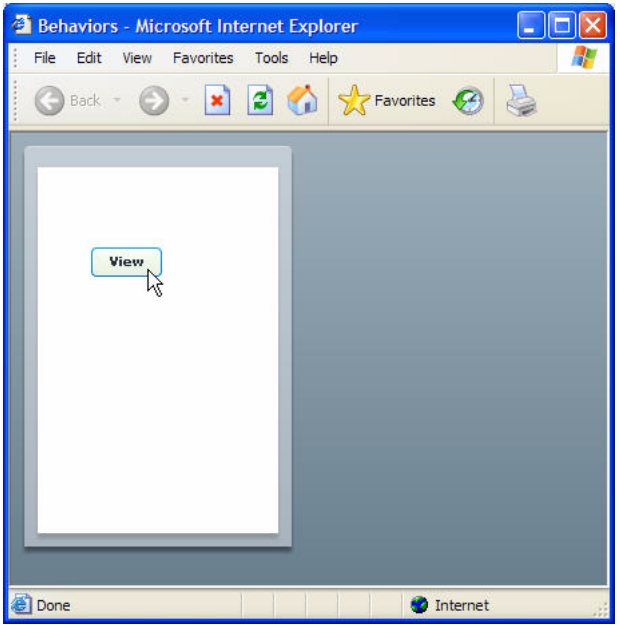
8. Save the file. The file should appear as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Glow id="buttonGlow" color="0x99FF66"
        alphaFrom="1.0" alphaTo="0.3"
        duration="1500"/>
    <mx:Panel x="10" y="10" width="200" height="300" layout="absolute">
        <mx:Button x="40" y="60" label="View" id="myButton"
            mouseUpEffect="{buttonGlow}"/>
    </mx:Panel>
</mx:Application>
```

Flex Builder compiles the application.

9. Click the Run button in the toolbar. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.

   A browser opens and runs your small Flex application. Click the View button. It should emit a green glow that gradually diminishes in intensity until it becomes a pale green.



Invoke an effect from a different component

Instead of component triggers, you can use Flex events to invoke effects. This ability lets you have one component invoke an effect that plays on a different component. For example, you can use a Button control's click event to instruct a TextArea control to play a Fade effect.

When the user clicks your application's View button, you want a Label component to appear with blurry text that gradually comes into focus to reveal a series of numbers.

1. In Design mode, insert a Label control below the View button and then set the following Label properties in the Properties view:
   - ID: **myLabel**
   - Text: **4 8 15 16 23 42**

- X: **40**
- Y: **100**

2. Switch to Source mode and define your Blur effect by entering the following `<mx:Blur>` tag after the `<mx:Glow>` tag:

```
<mx:Blur id="numbersBlur"
  blurYFrom="10.0" blurYTo="0.0"
  blurXFrom="10.0" blurXTo="0.0"
  duration="2000"/>
```

   The tag properties specify the starting and ending amounts of vertical and horizontal blur.

3. In the `<mx:Blur>` tag, specify the Label control called myLabel as the target of the effect (in bold):

```
<mx:Blur id="numbersBlur" target="{myLabel}"
  blurYFrom="10.0" blurYTo="0.0"
  blurXFrom="10.0" blurXTo="0.0"
  duration="2000"/>
```

   You want the effect to play on the component called myLabel.

4. In the `<mx:Button>` tag, specify the numbersBlur effect as the effect to play during a `click` event (in bold):

```
<mx:Button id="myButton" x="40" y="60" label="View"
  mouseUpEffect="{buttonGlow}" click="numbersBlur.play();"/>
```

   When a user clicks the Button control, the application invokes the effect by calling the effect's `play()` method.

   Because the numbersBlur effect targets the myLabel control, the application applies the effect to the label, not the button.

5. Hide the Label control from the user by setting its `visible` property to false in the `<mx:Label>` tag, as follows (in bold):

```
<mx:Label id="myLabel" x="40" y="100" text="4 8 15 16 23 42"
  visible="false"/>
```

   You don't want to display the numbers until the user clicks the View button.

6. Make the Label visible only when the user clicks the View button by programmatically setting its `visible` property to true in the button's `click` property, as follows (in bold):

```
<mx:Button id="myButton" x="40" y="60" label="View"
  mouseUpEffect="{buttonGlow}" click="numbersBlur.play();
  myLabel.visible=true;"/>
```

   When the user clicks the button, the blur effect starts playing and the Label becomes visible.

The Behaviors.mxml file should contain the following MXML code:

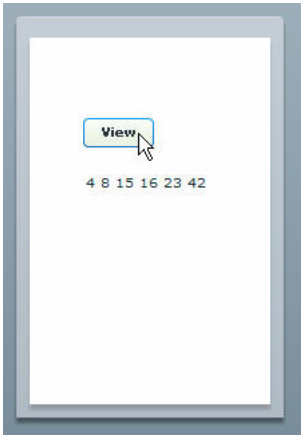
```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Glow id="buttonGlow" color="0x99FF66"
        alphaFrom="1.0" alphaTo="0.3"
        duration="1500"/>
    <mx:Blur id="numbersBlur" target="{myLabel}"
        blurYFrom="10.0" blurYTo="0.0"
        blurXFrom="10.0" blurXTo="0.0"
        duration="2000"/>
    <mx:Panel x="10" y="10" width="200" height="300" layout="absolute">
        <mx:Button x="40" y="60" label="View" id="myButton"
            mouseUpEffect="{buttonGlow}"
            click="numbersBlur.play(); myLabel.visible=true;"/>
        <mx:Label x="40" y="100" text="4 8 15 16 23 42" id="myLabel"
            visible="false"/>
    </mx:Panel>
</mx:Application>
```

7. Save the file.

   Flex Builder compiles the application.

8. Click the Run button in the toolbar.

   A browser opens and runs the application. Click the View button. The button emits a green glow while a series of blurred numbers gradually comes into focus.



Create a composite effect

You can make the Label component move down by 20 pixels while the numbers come into focus. In other words, you can combine your Blur effect with a Move effect.

Flex supports combining more than one effect to create a composite effect. You define a composite effect with either the `<mx:Parallel>` tag or the `<mx:Sequence>` tag, depending on whether you want the combined effects to play in parallel or sequentially. For your application, you want the Blur and Move effects to play in parallel.

1. In Source mode, start your composite effect by entering the following `<mx:Parallel>` tag before the `<mx:Blur>` tag:

   ```
   <mx:Parallel id="BlurMoveShow">
   </mx:Parallel>
   ```

   The name of your parallel composite effect is BlurMoveShow.

2. Select the full `<mx:Blur>` tag in your code, and then paste it between the opening and closing `<mx:Parallel>` tags so it becomes a child tag of the `<mx:Parallel>` tag.

3. Select the `target="{myLabel}"` property in the `<mx:Blur>` tag, and then move it into the opening `<mx:Parallel>` tag so it becomes a property of the `<mx:Parallel>` tag, as follows (in bold):

   ```
   <mx:Parallel id="BlurMoveShow" target="{myLabel}">
   ```

   You want the composite effect to play on the component called myLabel.

4. Define your new Move effect by entering the following `<mx:Move>` tag after the `<mx:Blur>` tag:

   ```
   <mx:Move id="numbersMove" yBy="20" duration="2000"/>
   ```

   You want the Label control to move down 20 pixels in two seconds.

   The completed `<mx:Parallel>` tag should look as follows:

   ```
   <mx:Parallel id="BlurMoveShow" target="{myLabel}">
     <mx:Blur id="numbersBlur"
       blurYFrom="10.0" blurYTo="0.0"
       blurXFrom="10.0" blurXTo="0.0"
       duration="2000"/>
     <mx:Move id="numbersMove" yBy="20" duration="2000"/>
   </mx:Parallel>
   ```

5. In the `<mx:Button>` tag, change the effect to play in response to the `click` event by replacing the numbersBlur effect with the BlurMoveShow composite effect, as follows (in bold):

   ```
   <mx:Button id="myButton" x="40" y="60" label="View"
     mouseUpEffect="{buttonGlow}" click="BlurMoveShow.play();
     myLabel.visible=true;"/>
   ```

6. Save the file. The final application should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Glow id="buttonGlow" color="0x99FF66"
        alphaFrom="1.0" alphaTo="0.3"
        duration="1500"/>
    <mx:Parallel id="BlurMoveShow" target="{myLabel}">
        <mx:Blur id="numbersBlur"
            blurYFrom="10.0" blurYTo="0.0"
            blurXFrom="10.0" blurXTo="0.0"
            duration="2000"/>
        <mx:Move id="numbersMove" yBy="20" duration="2000"/>
    </mx:Parallel>
    <mx:Panel x="10" y="10" width="200" height="300" layout="absolute">
        <mx:Button x="40" y="60" label="View" id="myButton"
            mouseUpEffect="{buttonGlow}"
            click="BlurMoveShow.play(); myLabel.visible=true;"/>
        <mx:Label x="40" y="100" text="4 8 15 16 23 42" id="myLabel"
            visible="false"/>
    </mx:Panel>
</mx:Application>
```

Flex Builder compiles the application.

7. Click the Run button in the toolbar.

   A browser opens and runs the application. Click the View button. The button emits a green glow while a series of blurred numbers gradually comes into focus while moving down 20 pixels.

In this lesson, you learned how to create behaviors, invoke an effect from a different component, and combine more than one effect to create a composite effect. To learn more, see Chapter 17, “Using Behaviors” in *Flex 2 Developer’s Guide*.

CHAPTER 14

Use View States and Transitions

14

You can use view states and transitions in Adobe Flex to create richer, more interactive user experiences. For example, you can use view states to create a user interface that changes its appearance based on the task the user is performing.

A *view state* is a named layout that you define for a single MXML application or component. You can define several view states for an application or component, and switch from one view state to another depending on the user's actions. View states allow you to dynamically change the user interface in response to users' actions or progressively reveal more information based on the context.

NOTE

Alternatively, you can use the ViewStack navigator container with other navigator containers to achieve similar results. For more information, see Chapter 16, "Using Navigator Containers" in *Flex 2 Developer's Guide*.

A *transition* is one or more effects grouped together to play when a view state changes. The purpose of a transition is to make the visual change smoother and more interesting.

This lesson shows you how to use view states and transitions to create a user interface that reveals more information when users request it.

In this lesson, you'll complete the following tasks:

Set up your project

Before you begin this lesson, ensure that you perform the following tasks:

- If you have not already done so, create the Lessons project in Adobe Flex Builder. See "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Design the base state

Before you can use view states, you must design the base state of the application or component. The base state is the default layout of the application or custom component.

In this section, you create a base state by inserting and positioning the controls of a simple search form.

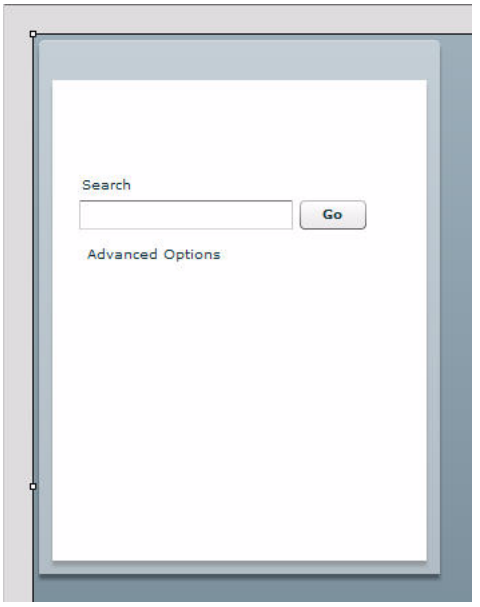
1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called ViewStates.mxml.

NOTE	For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the ViewStates.mxml file as the default file to be compiled by right-clicking (Control-click on Macintosh) the file in the Navigator view and selecting Set As Default Application from the context menu.
3. In the MXML editor's Design mode, add a Panel container to the ViewStates.mxml file by dragging one from the Layout category of the Components view (Window > Components).
4. Select the Panel container in the layout and set the following Panel properties in the Flex Properties view:
   - ID: **panel1**
   - Width: **300**
   - Height: **400**
   - X: 5
   - Y: 5

5. Add the following controls to the panel by dragging them into the panel from the Controls category of the Components view:
   - Label
   - TextInput
   - Button
   - LinkButton
6. Select the Label control in the panel and set the following Label properties in the Flex Properties view:
   - Text: **Search**
   - X: **20**
   - Y: **70**
7. Select the TextInput control and set the following TextInput properties in the Flex Properties view:
   - X: **20**
   - Y: **90**
8. Select the Button control and set the following Button properties in the Flex Properties view:
   - Label: **Go**
   - X: **185**
   - Y: **90**
9. Select the LinkButton control and set the following LinkButton properties in the Flex Properties view:
   - ID: **linkbutton1**
   - Label: **Advanced Options**
   - X: **20**
   - Y: **120**

The layout should look similar to the following example:



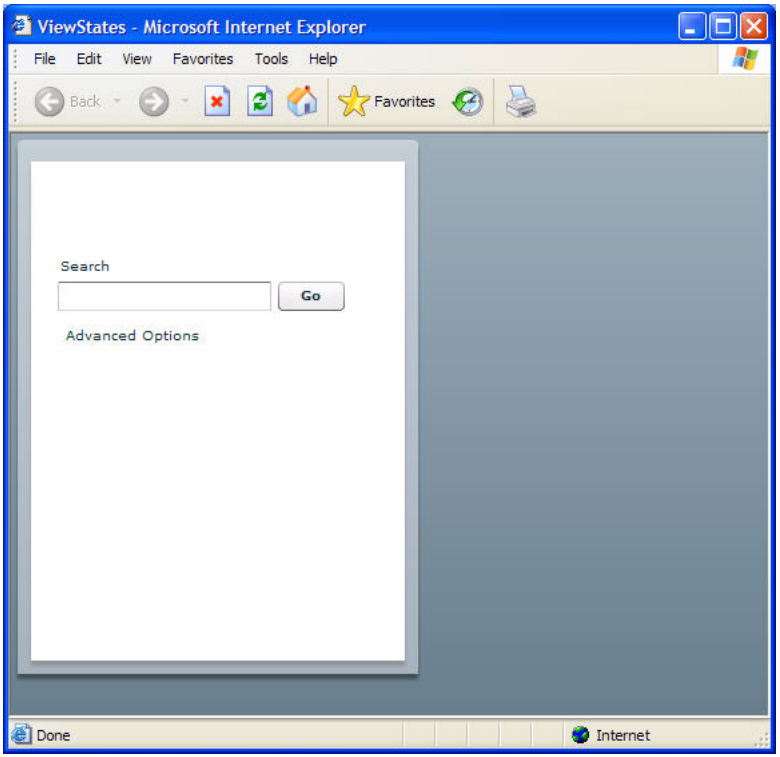
10. Switch to the editor's Source mode by clicking the Source button in the document toolbar.

The ViewStates.mxml file should contain the following MXML code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel id="panel1" x="5" y="5" width="300" height="400"
layout="absolute">
        <mx:Label x="20" y="70" text="Search"/>
        <mx:TextInput x="20" y="90"/>
        <mx:Button x="185" y="90" label="Go"/>
        <mx:LinkButton x="20" y="120" label="Advanced Options"
id="linkbutton1"/>
    </mx:Panel>
</mx:Application>
```

11. Save the file, wait until Flex Builder compiles the application, and click the Run button in the toolbar. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.

    A browser opens and runs the application.

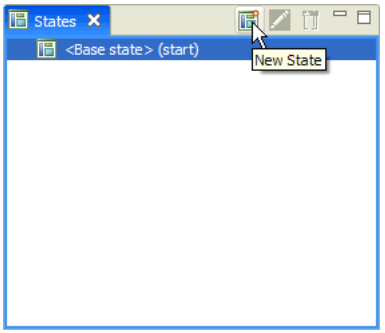


NOTE	The browser must have Flash Player 9 installed to run the application. You have the option of installing Flash Player 9 in selected browsers when you install Flex Builder. To switch to a browser with Flash Player 9, in Flex Builder select Window › Preferences › General › Web Browser.

Design a view state

The sample application provides a simple search mechanism that meets the needs of most users. However, some users might prefer to have more search options. You can use view states to provide these options on request.

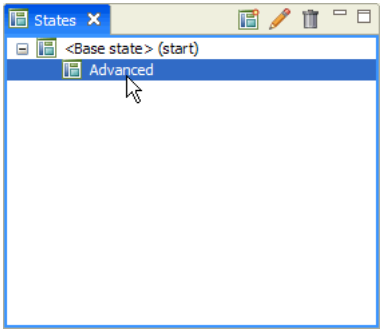
1. In Design mode, click the New State button in the States view (Window > States).



The New State dialog box appears.

2. Enter **Advanced** in the Name text box and click OK.

The new state appears in the States view.

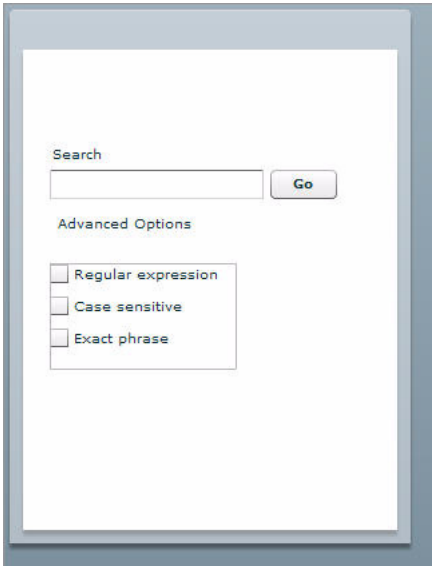


You can use the layout tools in Flex Builder to make changes to the appearance of the new state. You can modify, add, or delete components. As you work, the changes describing the new state are recorded in the MXML code.

3. In Design mode, insert a VBox container below the Advanced Search link, specifying a width of **160** and a height of **80** in the insert dialog box that appears, and then set the following VBox properties in the Flex Properties view:
   - ID: **myVBox**
   - X: **20**
   - Y: **160**
4. Drag three CheckBox controls into the VBox container.

   The VBox container automatically aligns the controls vertically.
5. Select the first CheckBox control in the VBox container and enter **Regular Expression** as the value of its `label` property.
6. Select the second CheckBox control and enter **Case sensitive** as the value of its `label` property.
7. Select the third CheckBox control and enter **Exact Phrase** as the value of its `label` property.

   The layout should look similar to the following:

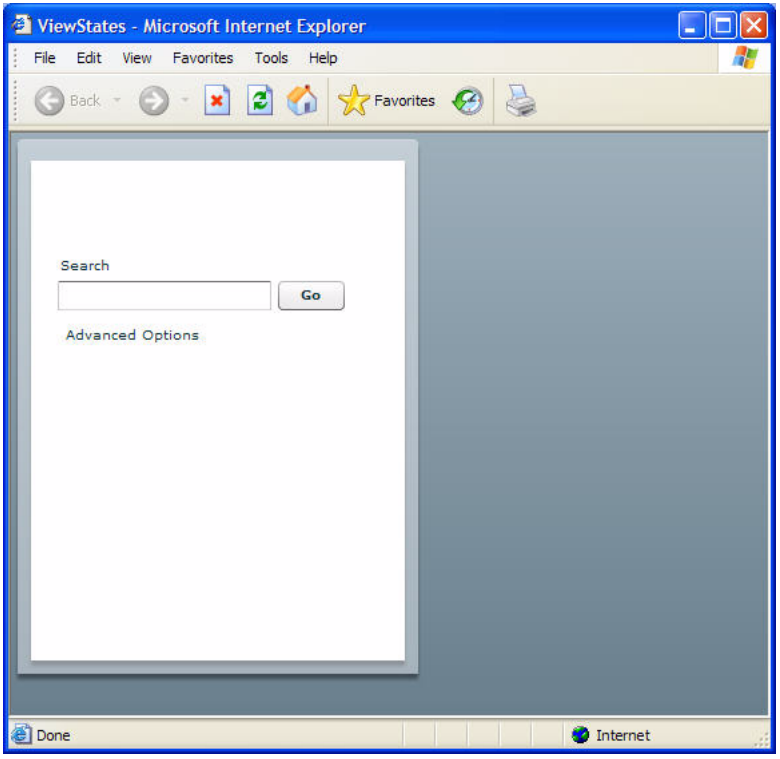


8. Switch to Source mode and examine the code.

   Flex Builder inserted a `<mx:states>` tag after the opening `<mx:Application>` tag, so that the application appears as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:states>
        <mx:State name="Advanced">
            <mx:AddChild relativeTo="{panel1}" position="lastChild">
                <mx:VBox x="20" y="160" width="160" height="80"
id="myVBox">
                    <mx:CheckBox label="Regular expression"/>
                    <mx:CheckBox label="Case sensitive"/>
                    <mx:CheckBox label="Exact phrase"/>
                </mx:VBox>
            </mx:AddChild>
        </mx:State>
    </mx:states>
    <mx:Panel id="panel1" x="5" y="5" width="300" height="400"
layout="absolute">
        <mx:Label x="20" y="70" text="Search"/>
        <mx:TextInput x="20" y="90"/>
        <mx:Button x="185" y="90" label="Go"/>
        <mx:LinkButton x="20" y="120" label="Advanced Options"
id="linkbutton1"/>
    </mx:Panel>
</mx:Application>
```

9. Save the file, wait until Flex Builder compiles the application, and click Run in the toolbar. A browser opens and runs the application.

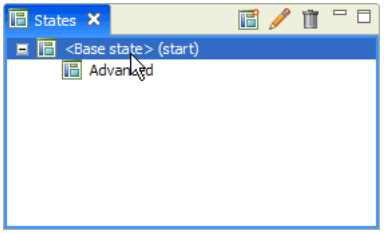


Your application does not display the controls you inserted in the new view state. By default, Flex applications display only the base state. You must define how users can switch view states, typically by clicking specific controls.

Define how users switch view states

You want to design your application so that when the user clicks the Advanced Options link, the layout switches to the Advanced view state with the extra search options. When the user clicks the link again, the layout switches back to the base state and hides the options. To do this, you need to define click event handlers to switch the states.

1. In Design mode, select the base state in the list in the States view.



   Because you will define a click event handler for the Link control that is part of the base state, you need to change the focus of the MXML editor to the base state.

   When you select the base state in this step, the MXML editor doesn't show the three CheckBox controls because they're not part of the base state.

2. Select the LinkButton control in the layout and enter the following value in the On Click text box in the Flex Properties view:

   ```
   currentState='Advanced'
   ```

   The `click` property specifies that when the user clicks the LinkButton control, the application should switch the current state to the Advanced view state, which displays the three additional check boxes.

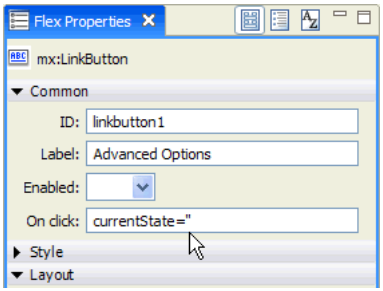
   Next, you want to hide the check boxes when the user clicks the LinkButton control a second time. You can do this by restoring the base state when the user clicks the link in the Advanced view state.

3. In the States view, select the Advanced state.

   Because you will define a click event handler for the LinkButton control when it's part of the Advanced state, you need to change the focus of the MXML editor to the Advanced state.

4. Select the LinkButton control in the layout of the Advanced view state, and then specify the following `click` property in the Flex Properties view:

```
currentState=''
```



Specify an empty string (two single quotes with no space between them) as the value of `currentState`. An empty string specifies the base state, so when the user clicks the LinkButton control in the Advanced view state, the base state is restored.

If you change to Source view, you will notice that Flex Builder added an `<mx:SetEventHandler>` tag to the application. The final application source code should look like the following:

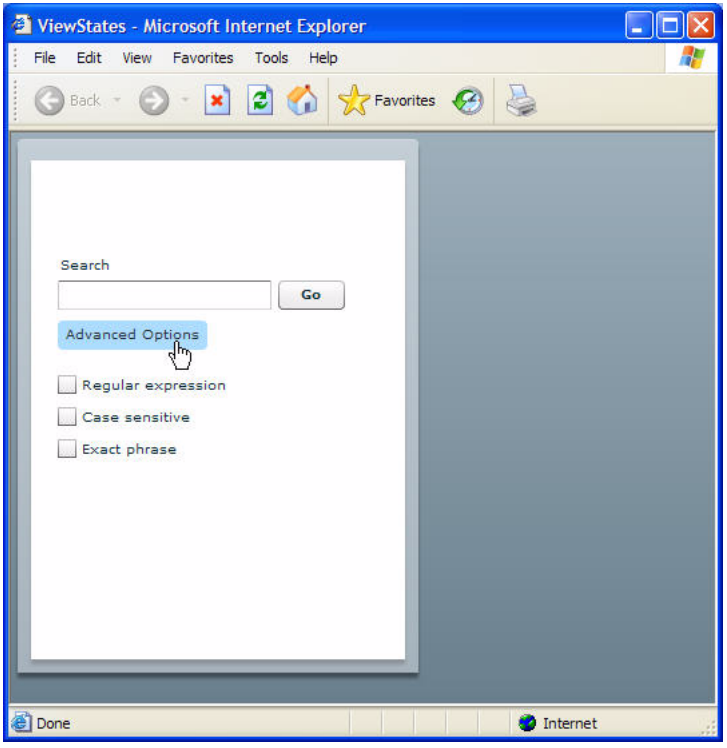
```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:states>
        <mx:State name="Advanced">
            <mx:AddChild relativeTo="{panel1}" position="lastChild">
                <mx:VBox x="20" y="160" width="160" height="80"
id="myVBox">
                    <mx:CheckBox label="Regular expression"/>
                    <mx:CheckBox label="Case sensitive"/>
                    <mx:CheckBox label="Exact phrase"/>
                </mx:VBox>
            </mx:AddChild>
            <mx:SetEventHandler target="{linkbutton1}" name="click"
handler="currentState=''"/>
        </mx:State>
    </mx:states>
    <mx:Panel id="panel1" x="5" y="5" width="300" height="400"
layout="absolute">
        <mx:Label x="20" y="70" text="Search"/>
        <mx:TextInput x="20" y="90"/>
        <mx:Button x="185" y="90" label="Go"/>
        <mx:LinkButton x="20" y="120" label="Advanced Options"
click="currentState='Advanced';" id="linkbutton1"/>
    </mx:Panel>
</mx:Application>
```

5. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button in the toolbar.

   A browser opens and runs the application.

6. Click the LinkButton control to view the advanced search options.

   The application displays the three check boxes you defined in the Advanced view state. The check boxes immediately appear on the screen.



7. Click the LinkButton control again to restore the base state, which hides the advanced search options.

Create a transition

When you change the view states in your application, the check boxes immediately appear on the screen. You can define a Flex transition that uses the WipeDown and Dissolve effects, which make the check boxes appear gradually rather than immediately.

1. In Source mode, create a Transition object and specify the change in state that triggers it by adding the following `<mx:transitions>` tag immediately after the closing `<mx:states>` tag:

```
<mx:transitions>
  <mx:Transition id="myTransition" fromState="*" toState="Advanced">
  </mx:Transition>
</mx:transitions>
```

   The code defines one Transition object called myTransition. (You can define more than one transition in the `<mx:transitions>` tag.)

   The code also specifies that the transition will be performed when the application changes from any view state (`fromState="*"`) to the Advanced view state (`toState="Advanced"`). The value `"*"` is a wildcard character specifying any view state.

2. Specify the targeted component for the transition, and how you want the effects to play—simultaneously or sequentially—by entering the following `<mx:Parallel>` tag (in bold) between the `<mx:Transition>` tags:

```
<mx:Transition id="myTransition" fromState="*" toState="Advanced">
  <mx:Parallel target="{myVBox}">
  </mx:Parallel>
</mx:Transition>
```

   The targeted component for the transition is the VBox container named myVBox.

   Because you want two effects to play simultaneously during the transition, you use the `<mx:Parallel>` tag. If you wanted the effects to play sequentially, you would use the `<mx:Sequence>` tag instead. In that case, the second effect would not start until the first effect finished playing.

3. Specify the effects to play during the transition by entering the following `<mx:WipeDown>` and `<mx:Dissolve>` tags (in bold) between the `<mx:Parallel>` tags:

```
<mx:Parallel target="{myVBox}">
  <mx:WipeDown duration="2000"/>
  <mx:Dissolve alphaFrom="0.0" alphaTo="1.0" duration="2000"/>
</mx:Parallel>
```

   You want to play two effects, a WipeDown effect that causes the targeted container to appear from top to bottom over a period of 2000 milliseconds, or 2 seconds, and a reverse Dissolve effect that causes the contents of the container to gradually come into focus in 2 seconds.

The completed application should look like the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:states>
        <mx:State name="Advanced">
            <mx:AddChild relativeTo="{panel1}" position="lastChild">
                <mx:VBox x="20" y="160" width="160" height="80"
id="myVBox">
                    <mx:CheckBox label="Regular expression"/>
                    <mx:CheckBox label="Case sensitive"/>
                    <mx:CheckBox label="Exact phrase"/>
                </mx:VBox>
            </mx:AddChild>
            <mx:SetEventHandler target="{linkbutton1}" name="click"
handler="currentState=''"/>
        </mx:State>
    </mx:states>
    <mx:transitions>
        <mx:Transition id="myTransition" fromState="*"
toState="Advanced">
            <mx:Parallel target="{myVBox}">
                <mx:WipeDown duration="2000"/>
                <mx:Dissolve alphaFrom="0.0" alphaTo="1.0"
duration="2000"/>
            </mx:Parallel>
        </mx:Transition>
    </mx:transitions>
    <mx:Panel id="panel1" x="5" y="5" width="300" height="400"
layout="absolute">
        <mx:Label x="20" y="70" text="Search"/>
        <mx:TextInput x="20" y="90"/>
        <mx:Button x="185" y="90" label="Go"/>
        <mx:LinkButton x="20" y="120" label="Advanced Options"
click="currentState='Advanced';" id="linkbutton1"/>
    </mx:Panel>
</mx:Application>
```

4. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button in the toolbar.

   A browser opens and runs the application.

5. Click the LinkButton control to view the advanced search options.

   The WipeDown and Dissolve effects play simultaneously, causing the advanced search options to appear gradually from top to bottom.

In this lesson, you used view states and transitions to create a more flexible user interface that provides users with more options on request. To learn more, see Chapter 6, "Adding View States and Transitions," on page 147 and the following chapters in *Flex 2 Developer's Guide*:

- Chapter 27, "Using View States"
- Chapter 28, "Using Transitions"

CHAPTER 15

Create a Custom Component

15

Creating custom MXML components can simplify the process of building complex applications. By breaking down the application into manageable chunks, you can write and test each component independently from the others. You can also reuse a component in the same application or in other applications, which saves development time.

This lesson shows you how to build an MXML component visually with Adobe Flex Builder and then how to insert it in an MXML application file.

In this lesson, you'll complete the following tasks:

Set up your project

Before you begin this lesson, ensure that you perform the following tasks:

- If you have not already done so, create the Lessons project in Flex Builder. For more information, see "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Create a test file for the custom component

You decide to build a login box as a custom MXML component. Before you start, however, you need to create an MXML application file to test it. An MXML application file is an MXML file that contains the `<mx:Application>` root tag. You can't compile and run an MXML component on its own; you must instead compile and run an MXML application file that uses the component.

In this section, you create an MXML application file to test your custom component.

1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Main.mxml.

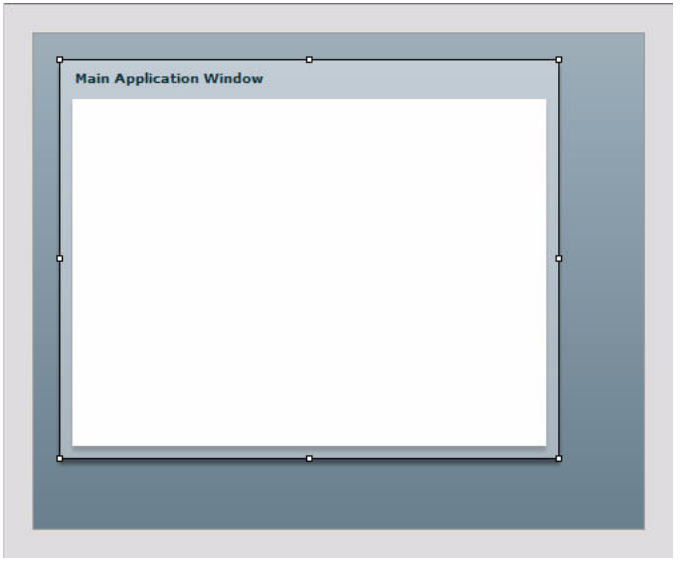
   NOTE

   For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the Main.mxml file as the default file to be compiled by right-clicking (Control-click on Macintosh) the file in the Navigator view and selecting Set As Default Application from the context menu.
3. In the MXML editor's Design mode, add a Panel container to the Main.mxml file by dragging it from the Layout category of the Components view (Window > Components).

   The Panel container is listed in the Layout category of components.
4. Select the Panel container in the Main.mxml file and set the following properties in the Flex Properties view:
   - Title: **Main Application Window**
   - Width: **375**
   - Height: **300**
   - X: **20**
   - Y: **20**

The layout should look similar to the following:



5. Save the file.

Now you can build and test your custom component.

Create the custom component file

The first step to building a custom MXML component is to create the file. Most custom components are derived from existing components. For your new login box component, you decide to extend the MXML Panel component because it provides a basic user interface for a login form.

Before you begin, create a subfolder to store the custom component files for your application.

1. In the Navigator view, right-click (Control-click on Macintosh) the Lessons parent folder and select New > Folder from the context menu.

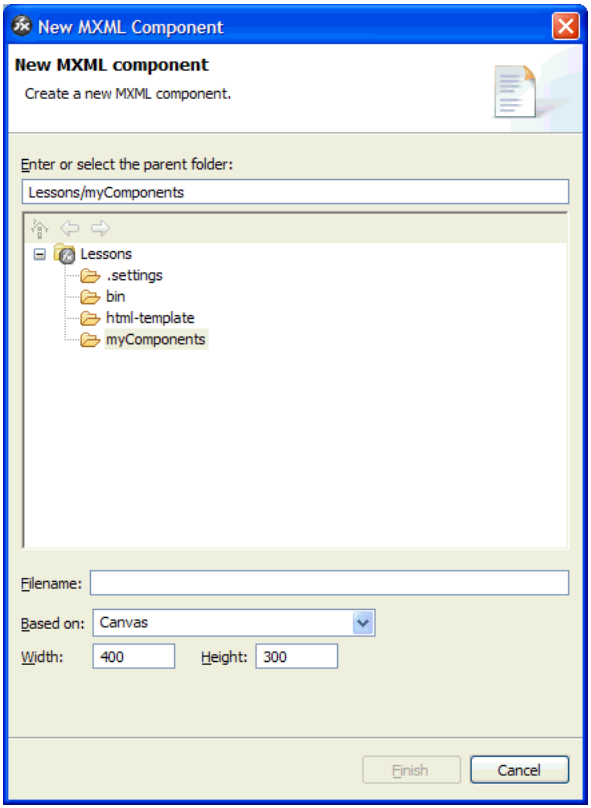
   The New Folder dialog box appears.

2. In the Folder Name text box, enter **myComponents** and click Finish.

   Flex Builder creates a new subfolder called myComponents.

3. With the myComponents folder still selected in the Navigator view, select File > New > MXML Component.

   The New MXML Component dialog box appears with the Lessons/myComponents folder set as the default folder for new custom components.



4. In the File Name text box, enter **LoginBox**.

   The filename also defines the component name.

5. In the Based On pop-up menu, select Panel.

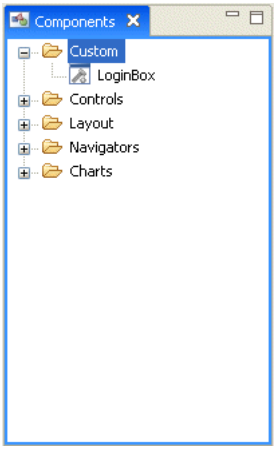
   You want to extend the Panel component.

6. In the Layout pop-up menu, make sure Absolute is selected (it should be the default).

7. Click Finish.

   Flex Builder creates and saves the LoginBox.mxml file in the myComponents folder and opens it in the MXML editor.

If you switch to Design mode, the component also appears in the Custom category of the Components view:



If you save a custom component file in the current project or in the source path of the current project, Flex Builder displays the component in the Components view so that you can rapidly insert it in your applications.

NOTE

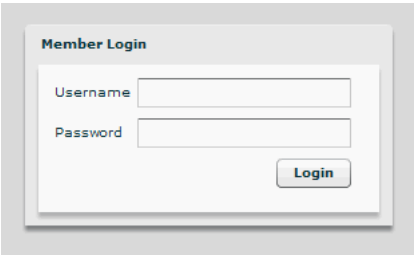
The Components view only lists visible custom components (components that inherit from UIComponent). For more information, see the *Adobe Flex 2 Language Reference*.

Design the layout of the custom component

The next step is to design the layout of the custom component. For your LoginBox component, you want a layout that includes username and password text boxes, and a submit button.

1. Make sure the LoginBox component is open in Design mode.
2. Select the Panel and set the following properties in the Flex Properties view:
   - Title: **Member Login**
   - Width: **275**
   - Height: **150**
3. Insert two Label controls in the panel and align them vertically.
4. Insert two TextInput controls to the right of the Label controls and align them vertically.
5. Select the first Label control and enter **Username** as the value of its `text` property.

6. Select the second Label control and enter **Password** as the value of its `text` property.
7. Select the first TextInput control and enter **txtUID** as the value of its `id` property.
8. Select the second TextInput control and enter **txtPwd** as the value of the ID text box and **true** as the value of the Display As Password text box.
9. Insert a Button control below the second TextInput control and enter **Login** as the value of its `label` property.
10. Align and fine-tune the position of the controls so that the layout looks as follows:



Your code should look as follows (your coordinate values may vary):

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
  width="275" height="150" title="Member Login">
  <mx:Label x="10" y="12" text="Username"/>
  <mx:Label x="10" y="42" text="Password"/>
  <mx:TextInput x="74" y="10" id="txtUID"/>
  <mx:TextInput x="74" y="40" id="txtPwd" displayAsPassword="true"/>
  <mx:Button x="178" y="70" label="Login"/>
</mx:Panel>
```

11. Save your file.

Define an event listener for the custom component

In some cases, you want the custom component to contain logic that can handle user actions. For your LoginBox component, you want the component to validate the user name and password when the user clicks the Login button, and then submit the data for authentication.

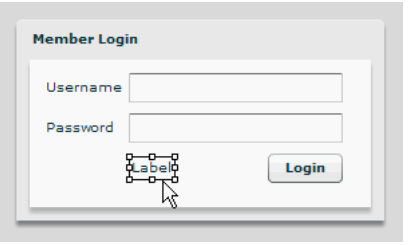
This section describes how to define a simple event listener for the Login button. An event listener is also known as an event handler. For a lesson on event listeners, see Chapter 12, "Use an Event Listener," on page 143.

NOTE

Developing the user authentication logic for the listener is outside the scope of this lesson.

First, you insert and modify a Label control to test that the event listener is called properly.

1. In Design mode, insert a Label control in the space to the left of the Login button, as follows:



2. Select the Label control and enter **lblTest** as the value of the Label's ID property, and clear the value of the Text property.
3. Select the Button control and enter **handleLoginEvent**() in the On Click text box in the Flex Properties view.

   When the user clicks the button, you want the application to call the `handleLoginEvent` function.

   Next, you write the listener function.
4. Switch to Source mode and place the insertion point immediately after the opening `<mx:Panel>` tag.
5. Start typing **<mx:Script>** until the full tag is selected in the code hints, press Enter to insert the tag in your code, and then type the closing angle bracket (>) to complete the tag.

   Flex Builder enters an `<mx:Script>` script block that also includes a CDATA construct.

NOTE

When using an <mx:Script> script block, you should wrap the contents in a CDATA construct. This prevents the compiler from interpreting the contents of the script block as XML, and allows the ActionScript to be properly generated.

6. Enter the following code in the CDATA construct:

```
private function handleLoginEvent():void {
  lblTest.text = "logging in...";
  //login logic
}
```

In a real application, the `handleLoginEvent` function would reference or contain the logic for validating and submitting the login entries for authentication. Developing the logic for the handler is outside the scope of this lesson.

The keyword `private` specifies the function's scope. In this case, the function is available only within the component. If you set the scope to `public`, the function is available throughout your code.

The keyword `void` specifies the function's return type. All ActionScript functions should specify a return type. The `handleLoginEvent` function returns no value.

The code for the component should look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
width="275" height="150" title="Member Login">
    <mx:Script>
        <![CDATA[
            private function handleLoginEvent():void {
                lblTest.text = "logging in...";
                //login logic
            }
        ]]>
    </mx:Script>
    <mx:Label x="10" y="12" text="Username"/>
    <mx:Label x="10" y="42" text="Password"/>
    <mx:TextInput x="74" y="10" id="txtUID"/>
    <mx:TextInput x="74" y="40" id="txtPwd" displayAsPassword="true"/>
    <mx:Button x="178" y="70" label="Login" click="handleLoginEvent()"/>
    <mx:Label x="74" y="72" id="lblTest"/>
</mx:Panel>
```

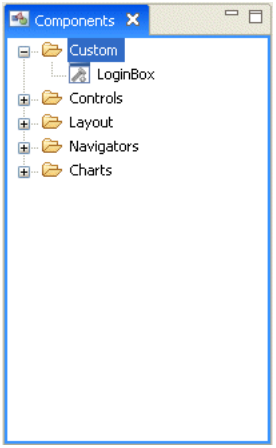
7. Save the file.

Use the custom component

The next step is to add the custom component to your MXML application file, and then to compile and run the application file to test the component.

1. In Design mode, switch to the Main.mxml file.

2. Locate the LoginBox component in the Custom category of the Components view.



3. Drag the LoginBox component into the panel in the layout.

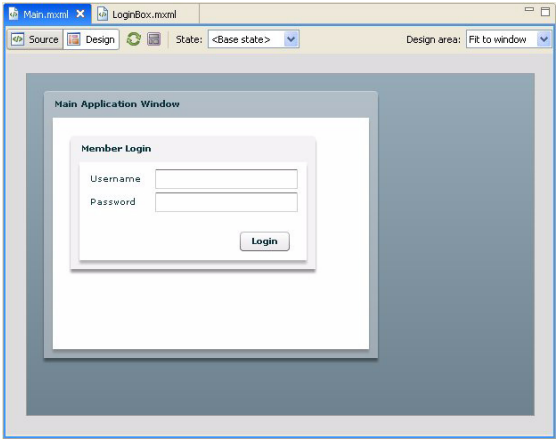
   Flex Builder inserts and renders the custom component in your layout like any other component.

4. With the LoginBox component still selected in the layout, set the following properties in the Properties view:

   - X: **20**
   - Y: **20**

   Flex Builder displays the properties of the custom component in the Properties view like any other component.

The layout should look similar to the following example:



5. Switch to MXML editor's Source mode by clicking the Source button in the editor's toolbar.

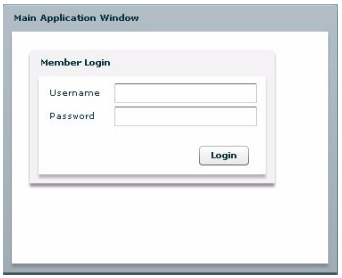
   Flex Builder inserted code in your application so that it looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute" xmlns:ns1="myComponents.*">
    <mx:Panel x="20" y="20" width="375" height="300" layout="absolute"
        title="Main Application Window">
    </mx:Panel>
    <ns1:LoginBox x="20" y="20">
    </ns1:LoginBox>
</mx:Application>
```

   When you dragged the custom component into the MXML file, Flex Builder defined a new namespace called ns1, and then inserted an `<ns1:LoginBox>` tag after the `<mx:Panel>` tag.

6. Save the file, wait until Flex Builder compiles the application, and click Run in the toolbar. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.

A browser opens and runs the application.



The application displays the LoginBox component you inserted in the main application file. You can reuse the same component in multiple MXML files.

Click the Login button to verify that the event listener is called properly. The string "logging in..." should appear to the left of the Login button.

In this lesson, you created a custom MXML component visually, and then used it in an MXML application file. You designed the component's layout and defined an event listener for a control in the component. To learn more, see Chapter 8, "Creating Custom MXML Components," on page 161 and Chapter 7, "Creating Simple MXML Components" in *Creating and Extending Flex 2 Components*.

CHAPTER 16

Use the Code Editor

16

This tutorial walks you through some of the essential code editing features in Adobe Flex Builder 2. Most of these features are available in both the MXML and ActionScript editors and learning how to use them will help to streamline your application development in Flex Builder.

For detailed information about the Flex Builder code editing features, see Chapter 9, "Code Editing in Flex Builder" in *Using Flex Builder 2*.

In this tutorial, you'll complete the following tasks:

Set up your project

Before you begin this lesson, perform the following tasks:

- If you haven't already done so, create the Lessons project in Flex Builder. For more information, see "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

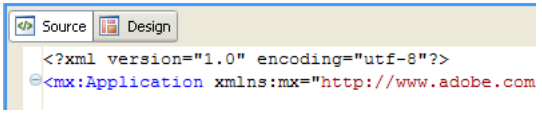
Create an MXML file to demonstrate code editing features

To demonstrate the features of the code editor, begin by creating an MXML application file in the Lessons project, which you'll use to add sample code.

1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called CodeEditor.mxml.

   NOTE: For the purpose of these short tutorials, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. By default, when you create an MXML application file, the editor is displayed in Source mode. If Design mode is displayed instead, select Source mode to continue.



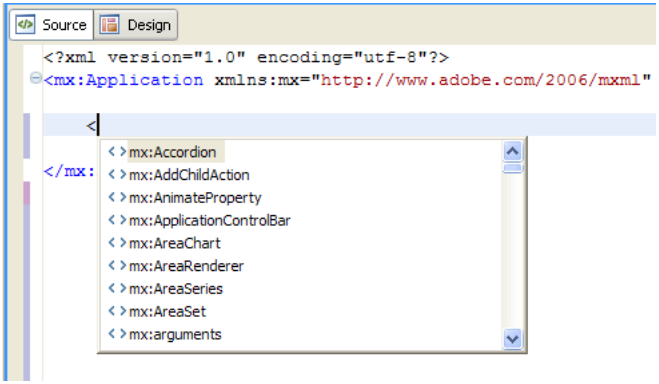
Use Content Assist

If you've walked through any of the other Flex Builder tutorials or if you've entered any code in the code editor, you will have noticed that Flex Builder automatically provides code hinting, which helps you to complete your code as you enter it.

1. With the CodeEditor.mxml file open in the editor, begin to add a Panel component by entering an opening tag:

   <

When you begin entering an MXML tag, code hints are immediately displayed, as shown here:



2. Type the word **panel** and you'll see that the list of options is reduced to display only those that contain the characters you have typed. Once you've entered the word panel, the Panel component is the only option displayed and you can press Enter to add it to your code.
3. To add attributes to the Panel component, press the Spacebar and code hints are once again displayed. The list of options is limited to those that can be added to the Panel component.
4. Enter the following line of code using the Content Assist feature:

   ```
   <mx:Panel width="250" height="250" layout="absolute" >
   ```

   When you enter the closing bracket, the closing tag is automatically added to your code:

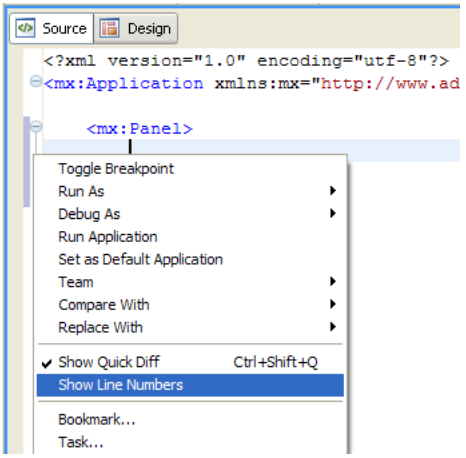
   ```
   <mx:Panel width="250" height="250" layout="absolute" >

   </mx:Panel>
   ```

Content Assist is available when editing MXML, ActionScript, and CSS code.

Show line numbers

To more easily locate lines of code, you can display line numbers in the editor.

- With the CodeEditor.mxml file open in the editor, right-click (Control-click on Macintosh) in the left margin of the editor and select Show Line Numbers, as shown here:



Line numbers are displayed in the editor.

Add a code comment

You can quickly add a comment or convert a block of your code into a comment, using a code editor keyboard shortcut.

1. In the code editor, position the cursor at a blank line and then press Control+Shift+C (Command+Shift+C on Macintosh). A comment tag is added to the document, as shown here:

   ```
   <!-- -->
   ```

2. Enter a comment between the opening and closing brackets, for example:

   ```
   <!-- This is a code comment in MXML -->
   ```

   In ActionScript, comment blocks are formatted using the standard ActionScript comment format, as shown here:

   ```
   /* This is a code comment in ActionScript */
   ```

3. To convert an existing block of code into a comment, select it and then press Control+Shift+C (Command+Shift+C on Macintosh). The entire selection is formatted as a comment.

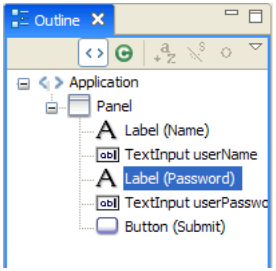
Use the Outline view

You use the Outline view to inspect the structure of and navigate to elements of your code.

1. Select and open the CodeEditor.mxml file and add the following code to the Panel or add any controls that you'd like:

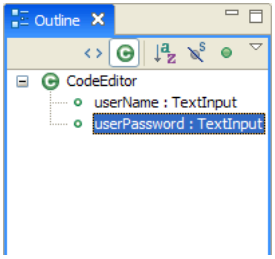
```
<mx:Panel width="250" height="250" layout="absolute" >
    <mx:Label text="Name" x="10" y="25"/>
    <mx:TextInput width="150" id="userName"  x="70" y="25"/>
    <mx:Label text="Password" x="10" y="55"/>
    <mx:TextInput width="150" id="userPassword" x="70" y="55"/>
    <mx:Button label="Submit"  x="80" y="150"/>
</mx:Panel>
```

2. Select the Outline view (Window > Outline or Window > Show Views > Flex > Outline in the plug-in configuration of Flex Builder) and you see the structure of your MXML document displayed as a hierarchy of components contained with the parent tag, as shown here:



3. Select a component (a Label for example) and it is highlighted in the code editor. Navigating your code using the Outline view is especially useful when you're working with large documents.

   The Outline view contains two modes: MXML and Class. In MXML mode, the Outline view displays the MXML components and their values (as shown above). In Class mode, the application is expressed as classes and their members, as shown here:



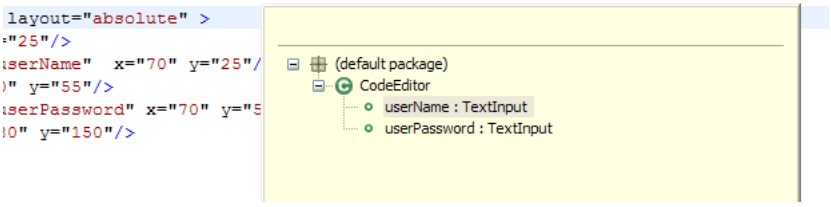
   Switch between the MXML and Class modes using the Outline view toolbar.

4. A quick way to locate and navigate to elements in the Outline view is to select the view and begin typing the name of an MXML component. Switch to the MXML view and then type **button**. The Button component is highlighted in the view and in the code editor.

Use the Quick Outline view in the code editor

Within the code editor itself, you can use the Quick Outline view, which you can access as you're entering code in the editor.

- Place the cursor anywhere within the code editor and press Control+O. The Quick Outline view is displayed in the code editor, as shown here:



The Quick Outline view displays your application's classes and their members. As in the Outline view, you can begin entering a class or member name to quickly locate and highlight the code.

Show language reference Help

As you enter code, you have quick access to the reference documentation for both Flex and ActionScript language elements.

1. In the sample code you created in a previous step, select the Label component, as shown here:

```
<mx:Panel width="250" height="250" layout="absolute" >
    <mx:Label text="Name" x="10" y="25"/>
    <mx:TextInput width="150" id="userName"  x="70" y="25"/>
```

2. Press Shift+F2.

   The Help system appears and the language reference topic for the Label component is displayed. Language reference topics contain properties, methods, and so on.

Open a code definition

To help simplify navigating to, inspecting, and modifying the various elements of your code that are located outside the document you are currently editing, you can open the source of an external code definition from where it is referred to in your code.

1. In the sample code contained in the CodeEditor.mxml file, select the Label component and press F3.
2. The file that contains the Label class code definition (part of the Flex framework) is opened in the editor and the Label class is highlighted.

NOTE	For the purpose of this short tutorial, we demonstrated opening the source of a code definition using one of the Flex framework components. The primary use of this feature is to interact with, for example, files that are shared between your projects such as ActionScript classes and MXML components. You should not attempt to edit the Flex framework classes.

In this lesson, you walked through a number of the key code editing features in Flex Builder. For more information about these and other code editing features, see Chapter 9, "Code Editing in Flex Builder" in *Using Flex Builder 2*.

CHAPTER 17

Debug an Application

17

This lesson shows you the basic steps of debugging your applications in Adobe Flex Builder. The debugging tools help you to monitor the state of your application when errors occur. You can then step line by line through your code, executing each line to inspect and change the value of local variables, method parameters, and component properties.

In this lesson, you'll complete the following tasks:

Set up your project

Before you begin this lesson, perform the following tasks:

- If you haven't already done so, create the Lessons project in Flex Builder. For more information, see "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Create an MXML file

To demonstrate the basic features of debugging your applications in Flex Builder, begin by creating an MXML application file in the Lessons project, which you'll use to add sample code.

1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Debugging.mxml.

   NOTE: For the purpose of these short tutorials, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

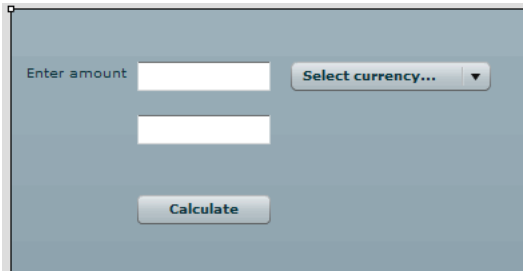
2. By default, when you create an MXML application file, the editor is displayed in Source mode. If Design mode is displayed instead, select Source mode to continue.
3. Designate the Debugging.mxml file as the default application file by right-clicking (Control-click on Macintosh) the file in the navigator view and selecting Application Management > Set As Default Application from the context menu.
4. In the MXML editor's Source mode, enter the following code after the opening `<mx:Application>` tag:

```
<mx:Label x="10" y="40" text="Enter amount"/>
<mx:TextInput id="txt_A"  x="95" y="40" width="100"/>
<mx:ComboBox x="210" y="40" id="cb_amount" dataProvider="{Currency}"
  prompt="Select currency..." width="150"/>
<mx:TextInput id="txt_B"  x="95" y="80" width="100" editable="false"/>
<mx:Text id="txt_currency" width="100"  x="210" y="80"/>
<mx:Button id="btn_calculate" click="calculate();" x="95" y="140"
  label="Calculate" width="100"/>
```

5. Save the file.

Preview the application in design view

The sample code you entered creates a simple currency converter that demonstrates the basic features of debugging applications in Flex Builder. To see what the sample application looks like, select the code editor's Design mode.



This simple application will calculate the exchange rate from one currency to another and display the conversion value.

Add a calculation function

An ActionScript function is used to capture the input from these controls, and calculate the exchange rate.

1. In the MXML editor's Source mode, enter the following public variable and function, so that your application looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Script>
        <![CDATA[
            [Bindable]
            public var Currency:Array = ["US Dollars", "Euro"];

            private function calculate():void {
                if (cb_amount.selectedItem == "US Dollars") {
                    var a:Number = Number(txt_A.text);
                    var c:Number = (a * 0.798072);
                    txt_B.text = String(c);
                    txt_currency.text = "Euro";
                } else if (cb_amount.selectedItem == "Euro") {
                    var b:Number = Number(txt_A.text);
                    var d:Number = (b * 1.25302);
                    txt_B.text = String(c);
                    txt_currency.text = "US Dollars";
                } else {
                    txt_currency.text = "Enter an amount and select a
currency";
                }
            }
        ]]>
    </mx:Script>
    <mx:Label x="10" y="40" text="Enter amount"/>
    <mx:TextInput id="txt_A"  x="95" y="40" width="100"/>
    <mx:ComboBox x="210" y="40" id="cb_amount" dataProvider="{Currency}"
prompt="Select currency..." width="150"/>
    <mx:TextInput id="txt_B"  x="95" y="80" width="100" editable="false"/
>
    <mx:Text id="txt_currency" width="100"  x="210" y="80"/>
    <mx:Button id="btn_calculate" click="calculate();" x="95" y="140"
label="Calculate" width="100"/>
</mx:Application>
```

The array variable provides selections for the ComboBox control and the Calculate function is called when the button is clicked.

This code sample purposely contains a logic error, which will help demonstrate how debugging works.

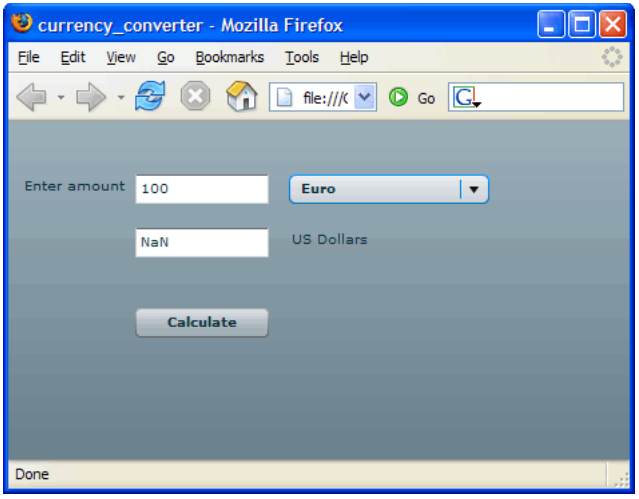
Run and test the application

Once you've entered the sample code, you can run and test the application.

1. Save the file and then click the Run command ( ) on the main toolbar. The application opens in a web browser.
2. To test the application, enter **100** as the amount, select US Dollars from the combo box control and then click the Calculate button.

   You should see the conversion amount displayed in euros.
3. Now, to demonstrate the flaw in the code, enter **100** as the amount, select Euro from the combo box, and then click the Calculate button.

   You should see the following:



   Rather than displaying a conversion value in US dollars, the `NaN` data type constant is displayed, indicating that the function has somehow failed.

   To help locate the flaw in the code, you use the debugging tools in Flex Builder.

Set a breakpoint

To debug the problem, you can set a breakpoint on the line of code to suspend the application so that you can inspect the function. Hitting a breakpoint activates the debugging perspective.

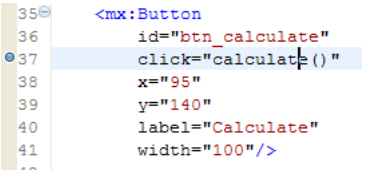
1. In the code editor, locate the line of code that contains the Button control. You'll see that the Button control's Click event is bound to the `calculate` function. Since this is where the function is called, this is where you set the breakpoint.

NOTE

When setting breakpoints in MXML it's important to understand that when you set a breakpoint on a line it is set for all elements of the line. If you set a breakpoint on this line and then run the application, the breakpoint will be activated as soon as the Button control is drawn on the screen, rather than when it is clicked. To work around this, you need to format your MXML to place a component's attributes on separate lines, as shown here:

```
<mx:Button
      id="btn_calculate"
      click="calculate();"
      x="95"
      y="140"
      label="Calculate"
      width="100"/>
```

2. Set a breakpoint on the line that contains the Click event handler by right-clicking (Control-click on Macintosh) in the left margin of the code editor. Select Toggle Breakpoint from the context menu and a breakpoint marker is added to the margin, as shown here:



3. Save the file and you're ready to debug the application.

Debug the sample application

Debugging an application means that you are running it in debug mode. This allows you to suspend the application when a breakpoint is encountered, inspect and modify the code, and then resume the application to continue debugging.

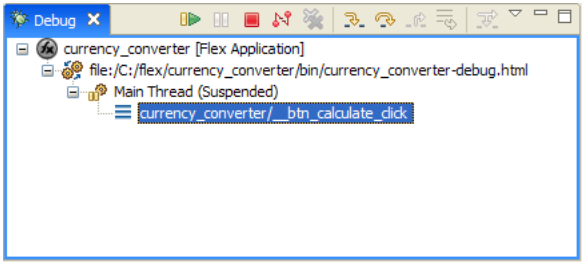
1. Click the Debug command ( ) on the main toolbar, or press F11.
2. Repeat the steps that created the error the first time you tested the application. Enter **100** as the amount, select Euro from the combo box, and then click the Calculate button
3. Since you set the breakpoint on the Calculate button, the application is suspended, which brings Flex Builder back into focus and switches you into the Debugging perspective.

   The line of code of containing the breakpoint is highlighted in the editor, as shown here:

```
33      <mx:TextInput id="txt_B"  x="95" y="80" width="100" editable="false"/>
34      <mx:Text id="txt_currency" width="100"  x="210" y="80"/>
35      <mx:Button
36          id="btn_calculate"
37          click="calculate()"
38          x="95"
39          y="140"
40          label="Calculate"
41          width="100"/>
42
43  </mx:Application>
```

   The Debugging perspective contains a number of views that provide the debugging tools that allow you to examine the state of the application, monitor important variables, management breakpoints and so on.

   The control center of the Debugging perspective is the Debug view, as shown here:



   The Debug view contains the commands that you use to control the debugging process, step you through the code, and then resume or stop running the application.

4. To begin debugging, you need to step into the function that is called by the button's `click` event. Select the Step Into command ( ) from the Debug view toolbar. The function is highlighted in the code editor, as shown here:

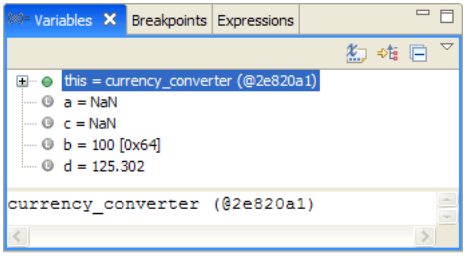
```
 8
 9          public function calculate():void {
10              if
11                  (cb_amount.selectedItem == "US Dollars")
12                  {
13                  var a:Number = Number(txt_A.text);
14                  var c:Number = (a * 0.798072);
15                  txt_B.text = String(c);
```

   Looking at the function you see that the troublesome section of code comes after the `else if` statement, where the calculation from euros into US dollars occurs.

5. Click the Step Into command again to step into the function. To navigate to the `else if` statement, click the Step Over command ( ). The Step Over command executes the current line of code and then suspends on the next line of code.

6. Continue clicking the Step Over command to execute each line of code.

   In the Variables view, the variables that are defined in the function are displayed. As each line of code is executed their values are calculated and displayed, as shown here:



   As you can see, the variable values are correct; multiplying 100 euros by the exchange rate produces the correct value (the amount entered is stored in the variable *b* and the result of the calculation is stored in variable *d*).

   Stepping through and executing each line of your code helps you understand what is and isn't working properly. In this case, since the values are correct, you can eliminate those lines of code as the source of the error and explore other possibilities.

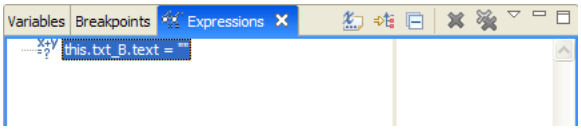
Watch a variable

Since the variables that store and calculate the data are correct, the next step is to examine the line of code that displays the result of the calculation, shown here:

```
txt_B.text = String(c);
```

Like the variables contained within the function, you can monitor the Text component's text property as you debug the application.

1. In the Variables view, select the `txt_B` text component (you may need to expand the `this` icon) and expand it to display its properties (which of course are variables in this context). Scroll down until you locate the `text` variable and select it.
2. Right-click (Control-click on Macintosh) to display the context menu and select Watch Variable.
3. Click the Expressions view and you'll see that the variable is now in the watch list.



4. Resume debugging and use the Step Over command to execute each line of code. After the line of code that is supposed to return the value of the conversion to dollars has been executed, you'll see that the text property of the text component has changed to '`NaN`'. This is the same incorrect result you got the first time you tested the application.

Correct the coding error

The coding error in the sample application occurs in the line of code that displays the value of the calculation from euros to US dollars. Variable *c* was used instead of variable *d*.

The corrected line of code looks like this:

```
txt_B.text = String(d);
```

If you run the application now, the correct value will be displayed.

In this lesson, you learned the basics of debugging in Flex Builder. While debugging can be complex, the debugging tools assist you inspecting your code and isolating the source of your errors. To learn more, see Chapter 11, "Running and Debugging Applications" in *Using Flex Builder 2*.

CHAPTER 18

Use Web Services

18

To provide data to your application, Adobe Flex includes components designed specifically for interacting with web services, HTTP servers, or remote object services (Java objects). These components are called remote procedure call (RPC) service components.

In this lesson, you create a simple reporting application for a blog aggregator that lists the most popular posts in the last 30 days. You decide to let the users determine the number of posts to list. You use an RPC service component called WebService to submit the user's selection and retrieve the data from a SOAP-based web service provided by the blog aggregator site, and then you display the data in a DataGrid control.

In this lesson, you'll complete the following tasks:

Set up your project

Before you begin this lesson, perform the following tasks:

- If you have not already done so, create the Lessons project in Adobe Flex Builder. For more information, see "Create the Lessons project" on page 101.
- Ensure that the automatic build option is enabled in Flex Builder. This option is enabled by default in the standalone configuration of Flex Builder but not in the plug-in configuration. To enable it, select Project > Build Automatically.

Review your access to remote data sources

For security reasons, applications running in Flash Player on client computers can only access remote data sources if one of the following conditions is met:

- Your application's compiled SWF file is in the same domain as the remote data source.
- You use a proxy and your SWF file is on the same server as the proxy.

  Adobe Flex Data Services provides a complete proxy management system for Flex applications. You can also create a simple proxy service using a web scripting language such as ColdFusion, JSP, PHP, or ASP. For more information on creating a proxy, see the following TechNote on the Adobe website at www.adobe.com/go/16520#proxy.
- A crossdomain.xml (cross-domain policy) file is installed on the web server hosting the data source.

  The crossdomain.xml file permits SWF files in other domains to access the data source. For more information on configuring crossdomain.xml files, see the following TechNote on the Adobe website at www.adobe.com/go/14213.

The data sources used in this lesson are located in a domain that has a crossdomain.xml setup. Therefore, Flash Player can access the remote data.

Review the API documentation

The MXNA blog aggregator provides a number of web services for developers at www.adobe.com/go/mxna_developers. Before you start building your application, you should review the API documentation for their web services to make sure a method exists that can retrieve the information you want. The API documentation for the web services is located at www.adobe.com/go/mxna_api.

The documentation describes a method called `getMostPopularPosts`. The method returns a number of posts with the most clicks in the last 30 days. For each post returned, the following information is provided: postId, clicks, dateTimeAggregated, feedId, feedName, postTitle, postExcerpt, postLink.

The method takes two required numeric parameters:

- `daysBack` specifies the number of days you want to go back.
- `limit` specifies the total number of posts you want returned; the number can't exceed 50.

With this information, you can use an RPC service component called WebService to consume the web service and retrieve the data you want—a list of the most popular posts in the last 30 days.

Insert and position controls

In this section, you create the layout of your reporting application. You decide to use a ComboBox control to let the users set the number of top posts to list, and a DataGrid to display the top posts.

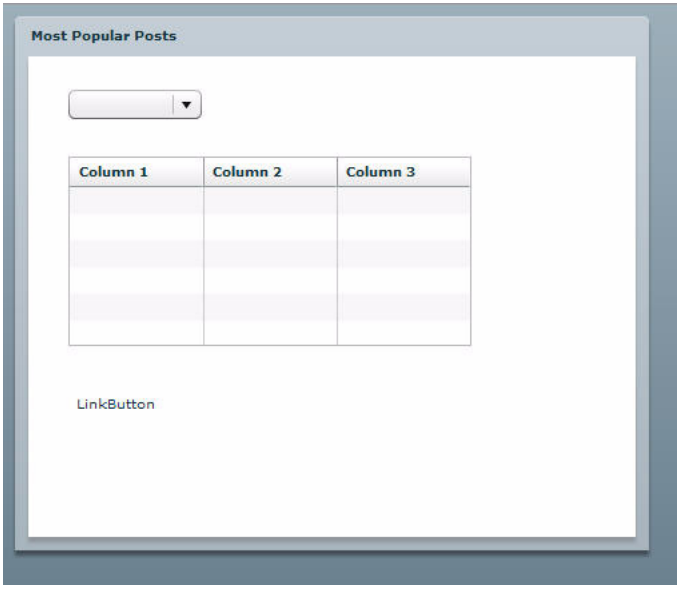
1. With your Lessons project selected in the Navigator view, select File > New > MXML Application and create an application file called Services.mxml.

   NOTE

   For the purpose of these lessons, several application files are used in a single Flex Builder project. However, it's good practice to have only one MXML application file per project.

2. Designate the Services.mxml file as the default file to be compiled by right-clicking (Control-click on Macintosh) the file in the Navigator view and selecting Set As Default Application from the context menu.
3. In the MXML editor's Design mode, drag a Panel container into the layout from the Layout category of the Components view, and then set the following Panel properties in the Properties view:
   - Title: **Most Popular Posts**
   - Width: 475
   - Height: **400**
   - X: **10**
   - Y: **10**

4. In Design mode, drag the following controls into the Panel container from the Components view:
   - ComboBox
   - DataGrid
   - LinkButton
5. Use the mouse to arrange the controls on the Canvas in a vertical, left-aligned column similar to the following:



6. Select the ComboBox control and enter **cbxNumPosts** as the value of its `id` property in the Flex Properties view.

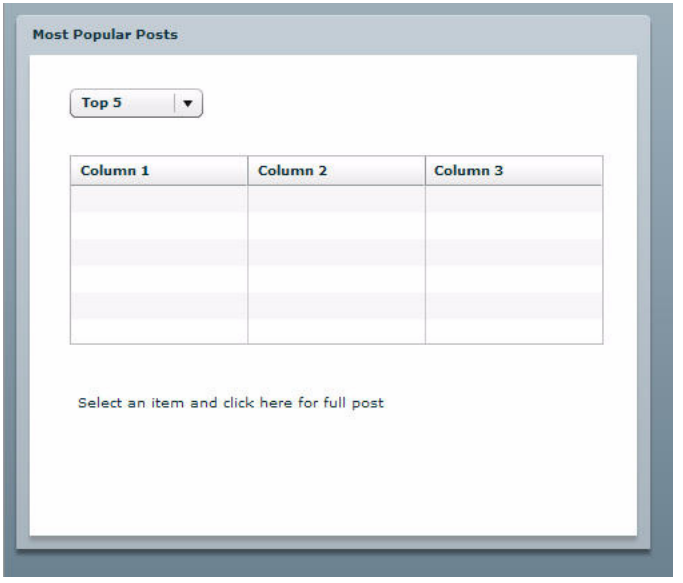
   The ComboBox control doesn't list any items. You populate the list next.
7. Switch to the editor's Source mode by clicking the Source button in the editor's toolbar, and then enter the following code between the opening and closing `<mx:ComboBox>` tag:

```
<mx:Object label="Top 5" data="5"/>
<mx:Object label="Top 10" data="10"/>
<mx:Object label="Top 15" data="15"/>
```

   For a tutorial on list-based controls, see Chapter 11, "Use List-based Form Controls," on page 135.

8. Switch back to Design mode, select the DataGrid component, and specify the following properties in the Flex Properties view:
   - ID: **dgTopPosts**
   - Width: **400**
9. Select the LinkButton control and enter **Select an item and click here for full post** as the value of its `label` property.

   The layout should look like the following example:



10. Switch to Source mode.

The Services.mxml file should contain the following MXML code (your coordinate values may vary):

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:Panel x="10" y="10" width="475" height="400" layout="absolute"
        title="Most Popular Posts">

        <mx:ComboBox x="30" y="25" id="cbxNumPosts">
            <mx:Object label="Top 5" data="5" />
            <mx:Object label="Top 10" data="10" />
            <mx:Object label="Top 15" data="15" />
        </mx:ComboBox>

        <mx:DataGrid x="30" y="75" id="dgTopPosts" width="400">
            <mx:columns>
                <mx:DataGridColumn headerText="Column 1"
dataField="col1"/>
                <mx:DataGridColumn headerText="Column 2"
dataField="col2"/>
                <mx:DataGridColumn headerText="Column 3"
dataField="col3"/>
            </mx:columns>
        </mx:DataGrid>

        <mx:LinkButton x="30" y="250"
            label="Select an item and click here for full post"/>
    </mx:Panel>

</mx:Application>
```

The next step is to insert and configure an RPC component called WebService in your application.

Insert a WebService component

You use the Flex WebService component to access a SOAP-based web service and retrieve information about recent blog posts.

1. In Source mode, enter the following `<mx:WebService>` tag immediately after the opening `<mx:Application>` tag:

```
<mx:WebService id="wsBlogAggr"
  wsdl="http://weblogs.macromedia.com/mxna/webservices/mxna2.cfc?wsdl"
  useProxy="false">
</mx:WebService>
```

The `wsdl` property specifies the location of the WSDL file for the web service. As of this writing, the location was still valid, but you should check to make sure it hasn't changed. You can find the latest WSDL file location on the developers page at www.adobe.com/go/mxna_developers, under Web Services.

NOTE

If you want, you can also use the following alias in the component: http://www.adobe.com/go/flex_mxna_wsdl.

The `useProxy` property specifies that you don't want to use a proxy on a server. For more information, see "Review your access to remote data sources" on page 206.

2. Specify the parameters to pass to the web service method.

   According to the API documentation, the `getMostPopularPosts` method takes the following required parameters:

   - `daysBack` specifies the number of days you want to go back.
   - `limit` specifies the total number of rows you want returned.

   To specify these parameters, enter the following tags between the opening and closing `<mx:WebService>` tags:

```
<mx:operation name="getMostPopularPosts">
  <mx:request>
    <daysBack>30</daysBack>
    <limit>{cbxNumPosts.value}</limit>
  </mx:request>
</mx:operation>
```

   The `name` property of an `<mx:operation>` tag must match the web service method name.

   You use a constant for the value of the `daysBack` parameter, but you bind the value of the `limit` parameter to the value of the selected item in the cbxNumPosts ComboBox control. You want the user to specify the number of posts to list.

   The next step is to prompt the application to call the web service method. You decide the method should be called when the ComboBox control changes in response to the user selecting an option.

3. In the `<mx:ComboBox>` tag, add the following `change` property (in bold):

```
<mx:ComboBox x="30" y="25" id="cbxNumPosts"
  change="wsBlogAggr.getMostPopularPosts.send()">
```

The final application code should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
layout="absolute">
    <mx:WebService id="wsBlogAggr"
        wsdl="http://weblogs.macromedia.com/mxna/webservices/
mxna2.cfc?wsdl"
        useProxy="false">
        <mx:operation name="getMostPopularPosts">
            <mx:request>
                <daysBack>30</daysBack>
                <limit>{cbxNumPosts.value}</limit>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:Panel x="10" y="10" width="475" height="400" layout="absolute"
        title="Most Popular Posts">

        <mx:ComboBox x="30" y="25" id="cbxNumPosts"
change="wsBlogAggr.getMostPopularPosts.send()">
            <mx:Object label="Top 5" data="5" />
            <mx:Object label="Top 10" data="10" />
            <mx:Object label="Top 15" data="15" />
        </mx:ComboBox>

        <mx:DataGrid x="30" y="75" id="dgTopPosts" width="400">
            <mx:columns>
                <mx:DataGridColumn headerText="Column 1"
dataField="col1"/>
                <mx:DataGridColumn headerText="Column 2"
dataField="col2"/>
                <mx:DataGridColumn headerText="Column 3"
dataField="col3"/>
            </mx:columns>
        </mx:DataGrid>

        <mx:LinkButton x="30" y="250"
            label="Select an item and click here for full post"/>
    </mx:Panel>

</mx:Application>
```

When the user selects an option in the ComboBox control, the `getMostPopularPosts` method of the wsBlogAggr WebService component is called. The method's parameters are specified in the WebService component's `<mx:operation>` tag. The `limit` parameter is set at run time depending on the option the user selects.

The application is ready to call the web service. The next step is to display the data returned by the web service.

Populate the DataGrid component

You want to use the DataGrid control to display the data returned by the web service. Specifically, you want to display the titles of the most popular posts and the number of clicks each has received.

1. In Source mode, enter the following `dataProvider` property in the `<mx:DataGrid>` tag (in bold):

   ```
   <mx:DataGrid x="30" y="75" id="dgTopPosts" width="400"
     dataProvider="{wsBlogAggr.getMostPopularPosts.lastResult}">
   ```

   You want to display the results of the web service's getMostPopularPosts operation in the DataGrid control.

2. In the first `<mx:DataGridColumn>` tag, enter the following `headerText` and `dataField` property values (in bold):

   ```
   <mx:DataGridColumn headerText="Top Posts" dataField="postTitle" />
   ```

   You want the first column in the DataGrid control to display the titles of the posts. You do this by identifying the field returned by the web service operation that contains the title data, and then entering the field name as the value of the `dataField` property. According to the API documentation for the `getMostPopularPosts` method, the field called `postTitle` contains the information you want.

3. In the second `<mx:DataGridColumn>` tag, enter the following `headerText`, `dataField`, and `width` property values (in bold):

   ```
   <mx:DataGridColumn headerText="Clicks" dataField="clicks" width="75" />
   ```

   You want the second column in the DataGrid control to display the number of clicks for each post during the last 30 days. According to the API documentation, the field that contains the data is called `clicks`.

4. Delete the third `<mx:DataGridColumn>` tag.

   You don't need a third column.

   The `<mx:DataGrid>` tag should look as follows:

   ```
   <mx:DataGrid x="30" y="75" id="dgTopPosts" width="400"
     dataProvider="{wsBlogAggr.getMostPopularPosts.lastResult}">
     <mx:columns>
       <mx:DataGridColumn headerText="Top Posts" dataField="postTitle"/>
       <mx:DataGridColumn headerText="Clicks" dataField="clicks"
     width="75"/>
     </mx:columns>
   </mx:DataGrid>
   ```

5. Save the file, wait until Flex Builder finishes compiling the application, and then click the Run button in the toolbar to test the application. If you're using the plug-in configuration of Flex Builder, select Run > Run As > Flex Application.

   A browser opens and runs the application. You find a problem in the application's default state. The ComboBox reads Top 5 but the DataGrid does not display any information.

   The DataGrid should display the top five posts, but it doesn't because your application hasn't called the web service yet. The application only calls it when the ComboBox changes. Even if you click Top 5 in the ComboBox after the application starts, the call is still not made because the selected item hasn't changed.

   To fix the problem, you decide to also call the web service immediately after the application is created, as follows.

6. In Source mode, enter the following `creationComplete` property (in bold) in the opening `<mx:Application>` tag:

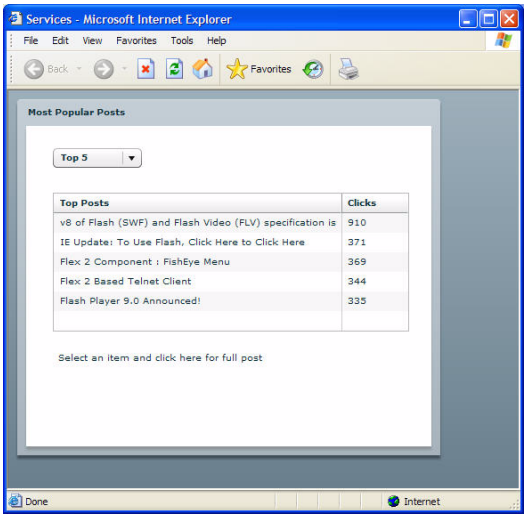
```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute"
  creationComplete="wsBlogAggr.getMostPopularPosts.send()">
```

The final application code should look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="absolute"
    creationComplete="wsBlogAggr.getMostPopularPosts.send()">
    <mx:WebService id="wsBlogAggr"
        wsdl="http://weblogs.macromedia.com/mxna/webservices/
mxna2.cfc?wsdl"
        useProxy="false">
        <mx:operation name="getMostPopularPosts">
            <mx:request>
                <daysBack>30</daysBack>
                <limit>{cbxNumPosts.value}</limit>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:Panel x="10" y="10" width="475" height="400" layout="absolute"
        title="Most Popular Posts">

        <mx:ComboBox x="30" y="25" id="cbxNumPosts"
change="wsBlogAggr.getMostPopularPosts.send()">
            <mx:Object label="Top 5" data="5" />
            <mx:Object label="Top 10" data="10" />
            <mx:Object label="Top 15" data="15" />
        </mx:ComboBox>

        <mx:DataGrid x="30" y="75" id="dgTopPosts" width="400"
dataProvider="{wsBlogAggr.getMostPopularPosts.lastResult}">
            <mx:columns>
                <mx:DataGridColumn headerText="Top Posts"
dataField="postTitle"/>
                <mx:DataGridColumn headerText="Clicks" dataField="clicks"
width="75"/>
            </mx:columns>
        </mx:DataGrid>

        <mx:LinkButton x="30" y="250"
            label="Select an item and click here for full post"/>
    </mx:Panel>

</mx:Application>
```

7. Save the file and run the application.

Blog titles and click statistics should appear in the DataGrid control after the application starts, confirming that the application successfully retrieved data from the web service and populated the control.



NOTE

There may be a delay of a few seconds before the data appears while the application is contacting the server.

Select another option from the ComboBox control to send a request to the web service and retrieve a longer or shorter list of posts.

Create a dynamic link

The web service doesn't provide the full text of the posts, but you still want users to be able to read the posts if they're interested. While the web service doesn't provide the information, it does provide the URLs to individual posts. According to the API documentation for the `getMostPopularPosts` method (see "Review the API documentation" on page 206), the information is contained in a field called `postLink`.

You decide to create a dynamic link that opens a browser and displays the full content of the post selected in the DataGrid control.

1. In Source mode, enter the following `click` property in the `<mx:LinkButton>` tag (in bold):

```
<mx:LinkButton x="30" y="250" label="Select an item and click here for
  full post"
  click="navigateToURL(new
  URLRequest(dgTopPosts.selectedItem.postLink));"/>
```

   The value of the link field of the selected item in the DataGrid control, `dgTopPosts.selectedItem.postLink`, is specified in the argument to the `navigateToURL()` method, which is called when the user clicks the LinkButton control. The `navigateToURL()` method loads a document from the specified URL in a new browser window.

NOTE	The navigateToURL() method takes a URLRequest object as an argument, which in turn takes a URL string as an argument.

2. Save the file, wait until Flex Builder finishes compiling the application, and click the Run button.

   A browser opens and runs the application. Click an item in the DataGrid control and then click the LinkButton control. A new browser window should open and display the blog page with the full post.

In this lesson, you used a WebService component to call and pass method parameters to a SOAP-based web service. You then bound the data returned by the web service to a DataGrid and a LinkButton control. To learn more, see the following topics in *Flex 2 Developer's Guide*:

- Chapter 44, "Understanding RPC Components"
- Chapter 45, "Using RPC Components"

CHAPTER 19

Use the Data Management Service

19

The Adobe Flex Data Management Service feature is a Flex Data Services feature that spans the client, network, and server tiers to provide distributed data in Flex applications. This tutorial provides two lessons about using the Data Management Service. The first lesson uses the ActionScript object Data Management Service adapter, which persists data in server memory and is useful for applications that require transient distributed data that is not persisted to a data store.

The second lesson uses the Data Management Service Java adapter for working with data that is persisted to a data store. The Java adapter passes data changes to methods available on an arbitrary Java class, referred to as the Java assembler. This adapter lets Java developers employ the Data Transfer Object (DTO) design pattern.

This tutorial provides the following lessons for building distributed applications that each use the Data Management Service, which is part of Flex Data Services:

Before you begin

Before you begin this tutorial, perform the following tasks:

- Ensure that you have installed Flex Data Services and that you can run the applications in the samples web application. Installation instructions are available at the following URL: www.adobe.com/go/flex2_installation
- Create a DSLessons directory that is a subdirectory of the samples directory of the samples web application.
- If you use Flex Builder for these lessons, do not use the Lessons project that you used for other lessons. Instead, create a new Flex Data Services project called DSLessons that uses the root directory of the samples web application as the root folder, and the context root of the samples web application as the root URL (for example, http://localhost:8700/samples/).

Build a distributed application with the ActionScript object adapter

In this lesson, you create a simple distributed application for entering and displaying notes that are shared among all clients. The application uses the ActionScript object adapter in Data Management Service to distribute text data to all clients. The ActionScript object adapter persists data in server memory and is useful for applications that require transient data that is not persisted to a permanent data store. Changes to data in one client are sent to the server-side Data Management Service and automatically propagated to other clients.

Configure a Data Management Service destination

In this section of the lesson, you define a server-side Data Management Service destination that uses the ActionScript object adapter. This destination persists data in server memory and distributes data to client applications.

1. In a text editor, open the data-management-config.xml file located in the WEB_INF/flex directory of the samples web application.

2. Directly above the text `<destination id="contact">`, make sure the following destination definition exists.

   Create the destination definition if it isn't there, and save the file.

```
<destination id="notes">
  <adapter ref="actionscript"/>
  <properties>
    <metadata>
      <identity property="noteId"/>
    </metadata>
  </properties>
</destination>
```

3. Start the application server that contains the samples web application.

Create an MXML file

In an MXML editor or text editor, create an MXML (text) file that contains the following text and save it as lesson1.mxml in the DSLessons directory of the samples web application.

As noted in "Before you begin" on page 220, if you are using Flex Builder, create this MXML file in a Flex Data Services project called DSLessons that uses the root directory of the samples web application as the root folder, the context root of the samples web application as the root URL (for example, http://localhost:8700/samples/), and the DSLessons directory as the project location.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
height="100%" width="100%">

</mx:Application>
```

If you are using Flex Builder, remove `layout="absolute"` from the `mx:application` tag and set lesson1.mxml as the Default Application.

Create the user interface

In this section, you create the TextArea control that displays editable text in the application.

1. Create a TextArea control by adding the following MXML code after the beginning `<mx:Application>` tag in the lesson1.mxml file.

```
  <mx:TextArea id="log" width="100%" height="80%"/>
  <mx:Button label="Send"/>
```

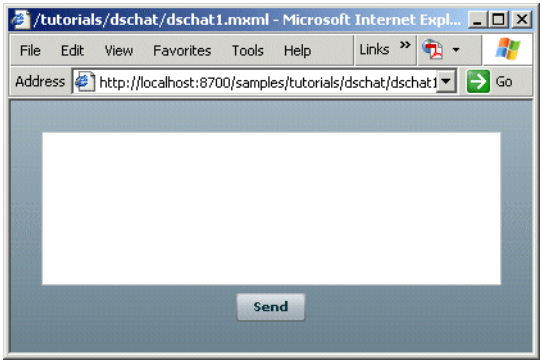
2. Save the file.

   If you are not using Flex Builder, open the following URL in a browser window:

   http://localhost:*port*/samples/DSLessons/lesson1.mxml

   If you are using Flex Builder, run the application from Flex Builder.

   The following application appears in the browser window:



Import the required ActionScript classes

In this section, you create a script block and import a set of classes that you will use within the script block.

1. Create a script block for ActionScript code directly below the `<mx:Application>` tag in the lesson1.mxml file:

```
<mx:Script>
  <![CDATA[

  ]]>
</mx:Script>
```

2. Directly below the `<![CDATA[` tag, add the following ActionScript import statements:

```
import mx.data.DataService;
import mx.data.events.*;
import mx.rpc.AsyncToken;
import mx.rpc.events.*;
import mx.messaging.events.*;
import mx.utils.ObjectProxy;
```

Create variables

In this section, in the script block you declare variables for objects that you will use.

Directly below the import statements in the script block, add the following variable declarations:

```
public var noteObj:Object = new Object();
public var getToken:AsyncToken;
private var ds:DataService;
[Bindable]
public var noteProxy:ObjectProxy;
```

Initialize the application

In this section, you write an event listener method that creates a DataService component and calls the DataService component's `getItem()` method, which retrieves the current note text.

1. Add the following method declaration directly under the variable declarations to create an event listener:

   ```
   public function initApp():void {

   }
   ```

2. Add the boldface text to the `initApp()` method to create a DataService component when the `initApp()` method is called. The `ds` DataService component connects to the server-side `notes` Data Management Service destination, which is specified in its one argument.

   ```
   public function initApp():void {
     ds = new DataService("notes");

   }
   ```

3. Add the boldface text to the `initApp()` method. This code adds a Result event listener to the DataService component and sets initial values for the noteObj object's `noteId` and `noteText` properties. It also sets the value of the getToken object to the AsyncToken object that the DataService component's `getItem()` method returns.

   The ActionScript object adapter uses the `noteId` property as a unique identifier for each noteObj object, and the `noteText` property contains the note text.

   ```
   public function initApp():void {
     ds = new DataService("notes");
     ds.addEventListener(ResultEvent.RESULT, resultHandler);
     ds.autoCommit = false;
     noteObj.noteId = 1;
     noteObj.noteText = "Type your notes here and share them with other
     clients!";
     getToken = ds.getItem(noteObj, noteObj);
   }
   ```

4. Add the boldface text to the `<mx:Application>` tag to call the `initApp()` method when the contact application is initialized:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="100%"
  width="100%" creationComplete="initApp();"
  height="100%" width="100%">
```

Send notes

In this section, you write an event listener method that sends text to the notes destination when you click the Send button. You then assign that method to the `click` event of the Send button.

1. Add the boldface text to create two-way binding between the `log.text` property and the `noteProxy.noteText` property:

```
<mx:Binding source="log.text" destination="noteProxy.noteText"/>
<mx:TextArea id="log" width="100%" height="80%"
  text="{noteProxy.noteText}"/>
```

2. Add the boldface text to the `<mx:Button>` tag to call the `sendMessage()` method when the Send button is clicked:

```
<mx:Button label="Send" click="ds.commit();"/>
```

Handle returned data

In this section, you write an event listener method that handles data that the Data Management Service returns to the client.

Add the following method just below the `initApp()` method:

```
public function resultHandler(event:ResultEvent):void
   {
      if (event.token == getToken)
          noteProxy = ObjectProxy(event.result);
   }
```

In the `resultHandler()` method, `event.token` is the AsyncToken that the `getItem()` method returns. Because the noteObj object is an anonymous object, the Data Management Service wraps it in an ObjectProxy object and returns that object in the result event.

Verify that your code is correct

Your code should match the following code example. Verify that the content is correct and save the lesson1.mxml file.

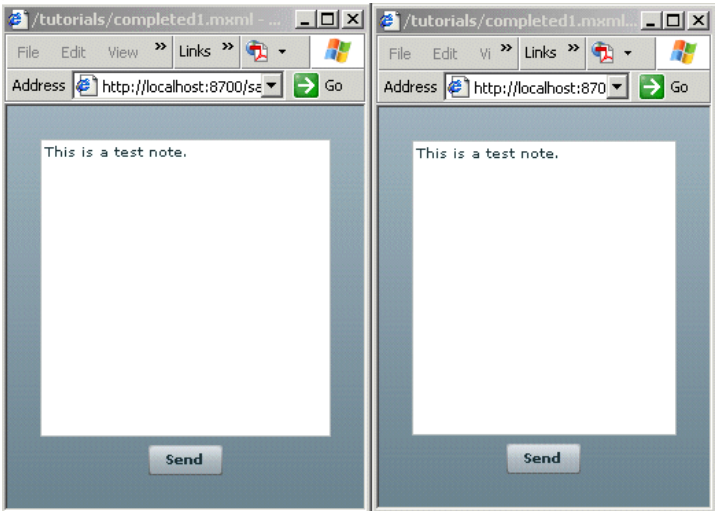
```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
 height="100%" width="100%"
 creationComplete="initApp();">

  <mx:Script>
    <![CDATA[
      import mx.data.DataService;
      import mx.data.events.*;
      import mx.rpc.AsyncToken;
      import mx.rpc.events.*;
      import mx.messaging.events.*;
      import mx.utils.ObjectProxy;

      public var noteObj:Object = new Object();
      public var getToken:AsyncToken;
      private var ds:DataService;
      [Bindable]
      public var noteProxy:ObjectProxy;

      public function initApp():void {
      ds = new DataService("notes");
      ds.addEventListener(ResultEvent.RESULT, resultHandler);
      ds.autoCommit = false;
      noteObj.noteId = 1;
      noteObj.noteText =
        "Type your notes here and share them with other clients!";
      getToken = ds.getItem(noteObj, noteObj);
   }
      public function resultHandler(event:ResultEvent):void {
        if (event.token == getToken)
        noteProxy = ObjectProxy(event.result);
      }
    ]]>
  </mx:Script>
  <mx:Binding source="log.text" destination="noteProxy.noteText"/>
  <mx:TextArea id="log" width="100%" height="80%"
    text="{noteProxy.noteText}"/>
  <mx:Button label="Send" click="ds.commit();"/>
</mx:Application>
```

Run the completed notes application

In this section, you run the completed notes application in two browser windows to see automatic updates in one window when data is changed in the other.

1. Run the application in two browser windows.
2. Make sure both instances of the application look like those in the following example.

   Resize the browser windows so that you can see both instances of the application at the same time.



3. In one of the browser windows, type a new line of text in the text area and click the Send button.

   Your change should update the application that is running in the other browser window.
4. In the second window, type additional text and click the Send button.

   The change should be appended to the text already displayed in the application that is running in the other browser window.

In this lesson, you used the Data Management Service feature with the ActionScript object adapter to create a distributed data application that automatically synchronized data among multiple clients and a server-side data resource. You built the client-side part of the application, and took a look at the server-side components of the application. To learn more, see the following topics in *Flex 2 Developer's Guide:*

- Chapter 50, "Understanding the Flex Data Management Service"
- Chapter 51, "Distributing Data in Flex Applications"
- Chapter 52, "Configuring the Data Management Service"

Build a distributed application with the Java adapter

In this lesson, you create a simple contact application that automatically retrieves contact information from a database and displays it in a DataGrid component. Changes to data in one client are sent to the server-side Data Management Service and automatically propagated to other clients.

After building and running the client application, you learn how the server-side Data Management Service enables the distribution and synchronization of data in the application.

In this tutorial, you'll complete the following tasks:

Install the contact application

The contact sample application used in this tutorial is not included in the Flex Data Services samples web application. The required files are available in the following zip file, which you can unzip into the samples (root directory) of the samples web application:

http://www.adobe.com/go/flex2_contact_app_zip

Copy the dataservice/contact/samples directory and its contents from the samples directory of the samples web application into the DSLessons directory that you created for this tutorial.

Copy the following destination definition after the last destination in the samples/WEB-INF/flex/data-management-config.xml file:

```
<destination id="contact">

  <adapter ref="java-dao" />

  <properties>
    <source>samples.contact.ContactAssembler</source>
    <scope>application</scope>

    <metadata>
    <identity property="contactId"/>
    </metadata>

    <network>
      <session-timeout>20</session-timeout>
      <paging enabled="false" pageSize="10" />
      <throttle-inbound policy="ERROR" max-frequency="500"/>
      <throttle-outbound policy="REPLACE" max-frequency="500"/>
    </network>

    <server>
      <fill-method>
      <name>loadContacts</name>
      </fill-method>

      <fill-method>
        <name>loadContacts</name>
        <params>java.lang.String</params>
      </fill-method>

      <sync-method>
        <name>syncContacts</name>
      </sync-method>
    </server>
  </properties>
</destination>
```

View the samples.contact.Contact class

The following example shows the code of the Contact.as file that is in the DSLessons/samples/contact directory. The `[RemoteClass]` metadata tag maps this class to a corresponding Java class on the server. The `[Managed]` metadata tag ensures that the managed Contact object supports the proper change events to propagate changes between the client-based object and the server-based object.

```
package samples.contact
{
  [Managed]
  [RemoteClass(alias="samples.contact.Contact")]
  public class Contact
  {
    public var contactId:int;

    public var firstName:String = "";

    public var lastName:String = "";

    public var address:String = "";

    public var city:String = "";

    public var state:String = "";

    public var zip:String = "";

    public var phone:String = "";
  }
}
```

Create an MXML file

In an MXML editor, create a text file that contains the following text and save it as lesson2.mxml in the DSLessons directory of the samples web application:

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

</mx:Application>
```

Create the user interface

In this section, you create the editable DataGrid control that displays editable contact information in the contact application.

1. Create a three-column editable DataGrid control by adding the following MXML code after the beginning `<mx:Application>` tag in the lesson2.mxml file.

   Set the `editable` property of the first column to `false`.

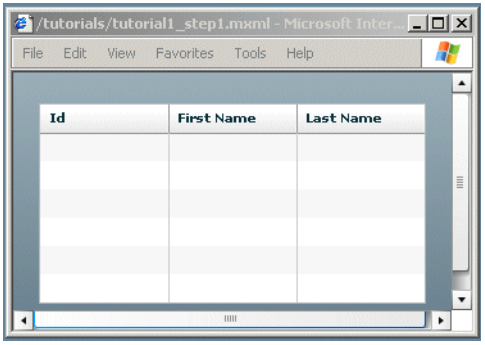
```
<mx:DataGrid id="dg" editable="true">
  <mx:columns>
    <mx:DataGridColumn dataField="contactId" headerText="Id"
  editable="false"/>
    <mx:DataGridColumn dataField="firstName" headerText="First Name"/>
    <mx:DataGridColumn dataField="lastName" headerText="Last Name"/>
  </mx:columns>
</mx:DataGrid>
```

2. Save the file.
3. If you are not using Flex Builder, open the following URL in a browser window:

   http://localhost:*port*/samples/DSLessons/lesson2.mxml

   If you are using Flex Builder, run the application from Flex Builder.

   The browser window should display the following application:



Import the required ActionScript classes

In this section, you import the mx.collections.ArrayCollection and mx.data.DataService classes so that you can create a DataService component and an ArrayCollection object. Later in the tutorial, you will create a DataService component that requests data from a server-side Data Management Service destination and fills an ArrayCollection object with that data.

1. Create a script block for ActionScript code directly above the `<mx:DataGrid>` tag in the lesson2.mxml file:

```
<mx:Script>
  <![CDATA[

  ]]>
</mx:Script>
```

2. Directly below the `<![CDATA[` tag, add the following ActionScript import statements:

```
import mx.data.DataService;
import mx.collections.ArrayCollection;
import samples.contact.Contact;
```

Create variables

In this section, you add variables for the ArrayCollection and DataService objects to the script block.

1. Directly below the import statements in the script block, add the following variable declarations for the `ds`, `contacts`, and `contact` variables:

```
public var ds:DataService;
public var contacts:ArrayCollection;
public var contact:Contact;
```

2. Add a `[Bindable]` metadata tag directly above the `contacts` variable declaration:

```
[Bindable]
public var contacts:ArrayCollection;
```

   The `[Bindable]` metadata tag indicates to the MXML compiler that `contacts` is a bindable property. By making it a bindable property, you can bind it into the `dataProvider` property of the DataGrid control and display data in the DataGrid control.

Bind the ArrayCollection object to the DataGrid

In this section, you bind the `contact` ArrayCollection object to the DataGrid control's `dataProvider` property to fill the DataGrid control with data from the ArrayCollection object.

1. Add the boldface text to the `<mx:DataGrid>` tag:

```
<mx:DataGrid id="dg" dataProvider="{contacts}" editable="true">
```

2. Save the lesson2.mxml file.

Fill the ArrayCollection object with data

In this section, you write an event listener method that creates a DataService object and an ArrayCollection object, and calls the DataService's `fill()` method to fill the ArrayCollection with data.

1. Add the following method declaration directly under the variable declarations to create an event listener:

```
public function initApp():void {

}
```

2. Add the boldface text to the initApp() method to create a DataService component and an ArrayCollection object when the `initApp()` method is called.

   The `ds` DataService connects to the server-side `contact` Data Management Service destination, which is specified in its one argument.

```
public function initApp():void {
  contacts = new ArrayCollection();
  ds = new DataService("contact");
}
```

3. Add the boldface text to the `initApp()` method to call the DataService object's `fill()` method when the `initApp()` method is called.

   The `fill()` method requests data from the server-side Data Management Service destination and fills the contacts ArrayCollection with that data. The DataService object manages all updates, additions, and deletions to the data in the ArrayCollection.

```
public function initApp():void {
  contacts = new ArrayCollection();
  ds = new DataService("contact");
  ds.fill(contacts);
}
```

4. Add the boldface text to the `<mx:Application>` tag to call the `initApp()` method when the contact application is initialized:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">
```

Verify that your code is correct

Your code should match the following code example. Verify that the content is correct and save the lesson2.mxml file.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  creationComplete="initApp();">

  <mx:Script>
    <![CDATA[
      import mx.data.DataService;
      import mx.collections.ArrayCollection;
      import samples.contact.Contact;
      public var ds:DataService;

      [Bindable]
      public var contacts:ArrayCollection;
      public var contact:Contact;

      public function initApp():void {
        contacts = new ArrayCollection();
        ds = new DataService("contact");
        ds.fill(contacts);
      }
    ]]>
  </mx:Script>

    <mx:DataGrid id="dg" dataProvider="{contacts}" editable="true">
      <mx:columns>
        <mx:DataGridColumn dataField="contactId" headerText="Id"
          editable="false"/>
        <mx:DataGridColumn dataField="firstName" headerText="First Name"
        />
        <mx:DataGridColumn dataField="lastName" headerText="Last Name"/>
      </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

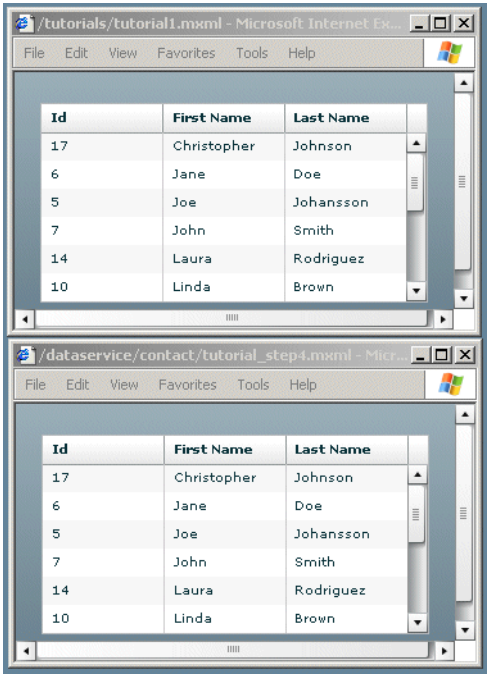
Run the completed contact application

In this section, you run the completed contact application in two browser windows to see automatic updates in one window when data changes in the other.

1. Run the application in two browser windows.

2. Ensure that both instances of the application look like the following example.

   Resize the browser windows so that you can see both instances of the application at the same time.



3. In one of the browser windows, click a DataGrid cell in the First Name column, and change the name.
4. Tab to the Last Name column of the DataGrid.

   Your change should automatically update the application that is running in the other browser window. After each change, the DataService component's `commit()` method is automatically called to send the changed data to the server-side Data Management Service.

The following sections explain some of the server-side functionality that enables data distribution and synchronization in the lesson2 application.

View the server-side Data Management Service destination

The lesson2.mxml file contains a DataService component that takes the name of a server-side Data Management Service destination named `contact` as an argument in its constructor. This reference to the contact destination is the only thing the client application needs to communicate with the destination. The `contact` destination is defined in the data-management-config.xml file in the WEB_INF/flex directory of the samples web application. The following example shows the XML code that defines the destination:

```
<destination id="contact">
   <adapter ref="java-dao" />
   <properties>
      <source>samples.contact.ContactAssembler
      </source>
      <scope>application</scope>
      <metadata>
         <identity property="contactId"/>
      </metadata>
      <network>
         <session-timeout>20</session-timeout>
         <paging enabled="false" pageSize="10"/>
         <throttle-inbound policy="ERROR" max-frequency="500"/>
         <throttle-outbound policy="REPLACE" max-frequency="500"/>
      </network>
      <server>
         <fill-method>
            <name>loadContacts</name>
         </fill-method>
         <fill-method>
            <name>loadContacts</name>
            <params>java.lang.String</params>
         </fill-method>
         <sync-method>
            <name>syncContacts</name>
         </sync-method>
      </server>
   </properties>
</destination>
```

Reference to a Data Management Service adapter configuration; this destination uses the Java object adapter

Java class that carries out data fill and synchronization operations between client and server.

Network -related settings

Mapping to method on adapter class to get data.

Mapping to method on adapter class that synchronizes multiple versions of data.

The previous example calls out the following elements of the `contact` destination definition:

Destination section	Description
`adapter`	References the Java object adapter configuration that the contact destination uses. This is also defined in the data-management-config.xml file. The Java object adapter lets you interact with a Java object to obtain data and commit data to a server-side data resource.
`source`	Specifies the assembler class, which is the Java class that passes data between a server-side data resource and the client-side DataService component. The assembler class is a custom class that you write. It is usually an implementation of the Transfer Object Assembler design pattern, which is described at http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html. The `scope` element below the `source` element indicates the scope of the class; valid values are `application`, `session`, and `request`.
`network`	Contains settings for how data messages are passed between the server-side Data Management Service and the client-side DataService component. For example, the `paging` element specifies whether data sent from the server to the client is chunked into smaller subsets of data instead of being sent all at once.
`fill-method`	Specifies a mapping to a fill method that is invoked when the client-side `DataService.fill()` method is called to fill an ArrayCollection object. You can implement any number of methods as fill methods, but they must be differentiated by the types of their parameters. Each of these methods can accept an arbitrary number of parameters of varying types. Based on the parameters that the client-side `DataService.fill()` method provides, the appropriate fill method on the assembler class is invoked.
`sync-method`	Specifies a method that accepts a list of data changes to allow synchronization of data among multiple clients and the backend data resource. A sync method takes a single argument, which is a standard java.util.List implementation that contains objects of type flex.data.ChangeObject. Each ChangeObject object in the list contains methods to access the application-specific changed Object instance, as well as convenience methods for describing the type of change and for accessing the changed data members.

View the assembler class

The contact destination uses the Java adapter. This is one of several types of Data Management Service adapters that Flex Data Services provides. As previously noted, the contact destination specifies an assembler class, which is a custom Java class that gets data from a data resource and handles the synchronization of data among clients and the data resource.

An assembler must have a zero-argument constructor. The assembler for this application is instantiated in the application scope, which means that the entire web application has only one instance of the class. The `scope` element in the `properties` section of the destination specifies that the assembler in the application scope.

The destination specifies the methods of the assembler class that are invoked to get data and synchronize multiple versions of data. In addition to those methods, the assembler class also implements methods for getting individual data items, and creating, updating, and deleting data items; these methods are implementations of methods in the flex.data.ChangeObject interface.

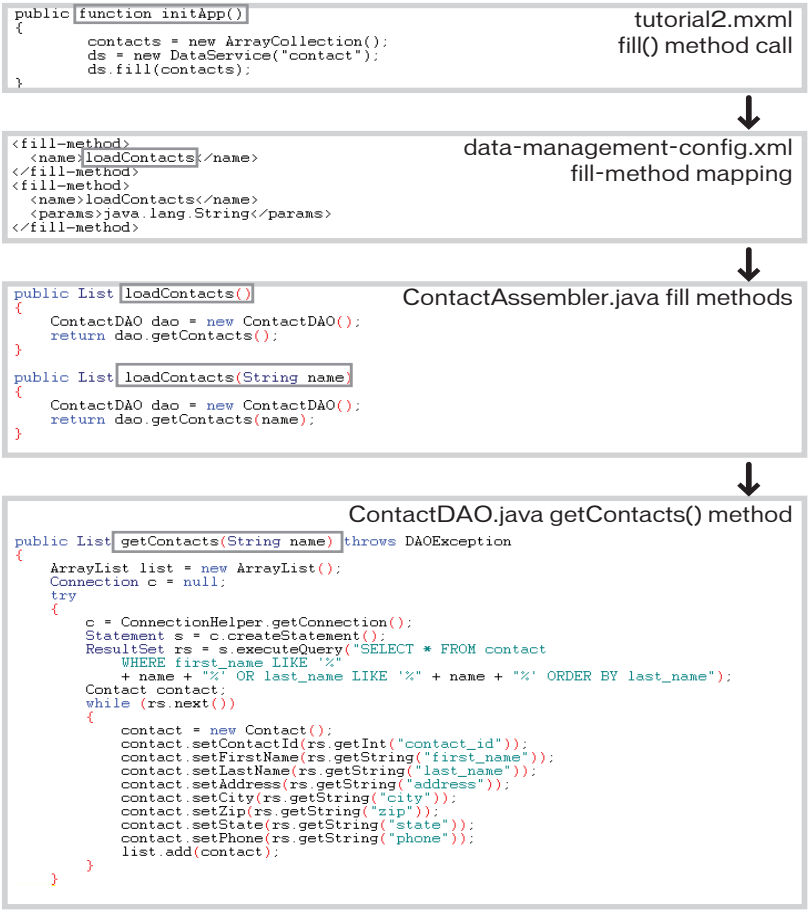
The following example shows the source code of the contact application's assembler class. This class delegates the actual calls to a SQL database, to a data access object (DAO) called ContactDAO. The source code for the ContactAssembler class and the ContatcDAO class are in the WEB_INF/src/samples/contact directory. The compiled classes are in the WEB_INF/classes/samples/contact directory.

View the fill methods

A fill method of the assembler class is called as a result of the client-side DataService's `fill()` method being called. The following code example shows the two methods that are specified as fill methods in the contact destination definition. The methods have the same name but different signatures, yet both return a List object that contains Contact objects. One method takes no arguments, while the other takes a name of type String as an argument. One of the methods is called, depending on whether or not the `DataService.fill()` request from the client specifies an argument of type String.

```
...
import flex.data.ChangeObject;
...
public class ContactAssembler {
...
  public List loadContacts() {
    ContactDAO dao = new ContactDAO();
    return dao.getContacts();
  }

  public List loadContacts(String name) {
    ContactDAO dao = new ContactDAO();
    return dao.getContacts(name);
  }
...
```

The following example shows the flow of a `fill()` method call from the client-side contact application:




```
public function initApp()
{
        contacts = new ArrayCollection();
        ds = new DataService("contact");
        ds.fill(contacts);
}
```

```
<fill-method>
  <name>loadContacts</name>
</fill-method>
<fill-method>
  <name>loadContacts</name>
  <params>java.lang.String</params>
</fill-method>
```

```
public List loadContacts()
{
    ContactDAO dao = new ContactDAO();
    return dao.getContacts();
}

public List loadContacts(String name)
{
    ContactDAO dao = new ContactDAO();
    return dao.getContacts(name);
}
```

```
public List getContacts(String name) throws DAOException
{
    ArrayList list = new ArrayList();
    Connection c = null;
    try
    {
        c = ConnectionHelper.getConnection();
        Statement s = c.createStatement();
        ResultSet rs = s.executeQuery("SELECT * FROM contact
            WHERE first_name LIKE '%"
            + name + "%' OR last_name LIKE '%" + name + "%' ORDER BY last_name");
        Contact contact;
        while (rs.next())
        {
            contact = new Contact();
            contact.setContactId(rs.getInt("contact_id"));
            contact.setFirstName(rs.getString("first_name"));
            contact.setLastName(rs.getString("last_name"));
            contact.setAddress(rs.getString("address"));
            contact.setCity(rs.getString("city"));
            contact.setZip(rs.getString("zip"));
            contact.setState(rs.getString("state"));
            contact.setPhone(rs.getString("phone"));
            list.add(contact);
        }
    }
```

View the sync method

The sync method of an assembler class lets you handle data changes sent from client-side DataService components. A sync method accepts one input parameter, which is a java.util.List object that contains a list of data changes of type flex.data.ChangeObject. The list of changes can include new data items, updates, and deletions.

Depending on whether a change is an add, update, or delete, the sync method calls the class's `doCreate()`, `doUpdate()`, or `doDelete()` method. The `doCreate()`, `doUpdate()`, and `doDelete()` methods are implementations of methods in the flex.data.ChangeObject interface. These methods call methods on the ContactDAO object, which interacts with a SQL database.

The following example shows the Java source code for the ContactAssember class's sync method:

```
...
import flex.data.ChangeObject;
...
public class ContactAssembler {
...
  public List syncContacts(List changes) {
    Iterator iterator = changes.iterator();
    ChangeObject co;
    while (iterator.hasNext()) {
      co = (ChangeObject) iterator.next();
      if (co.isCreate()) {
        co = doCreate(co);
      }
      else if (co.isUpdate()) {
        doUpdate(co);
      }
      else if (co.isDelete()) {
        doDelete(co);
      }
    }
    return changes;
   }
```

The following example shows the assembler class's `doCreate()`, `doUpdate()`, and `doDelete()` methods:

```
  private ChangeObject doCreate(ChangeObject co)
  {
    ContactDAO dao = new ContactDAO();
    Contact contact = dao.create((Contact) co.getNewVersion());
    co.setNewVersion(contact);
    return co;
  }

  private void doUpdate(ChangeObject co)
  {
    ContactDAO dao = new ContactDAO();
    try
    {
      dao.update((Contact) co.getNewVersion(), (Contact)
  co.getPreviousVersion());
    }
    catch (ConcurrencyException e)
    {
      System.err.println("*** Throwing DataSyncException when trying to
  update contact id=" + ((Contact) co.getNewVersion()).getContactId() );
      throw new DataSyncException(co);
    }
  }

  private void doDelete(ChangeObject co)
  {
    ContactDAO dao = new ContactDAO();
    try
    {
      dao.delete((Contact) co.getPreviousVersion());
    }
    catch (ConcurrencyException e)
    {
      System.err.println("*** Throwing DataSyncException when trying to
  delete contact id=" + ((Contact) co.getNewVersion()).getContactId() );
      throw new DataSyncException(co);
    }
  }
```

In this lesson, you used the Data Management Service feature with the Java adapter to create a distributed data application that automatically synchronized data among multiple clients and a server-side data resource. You built the client-side part of the application, and took a look at the server-side components of the application. To learn more, see the following topics in *Flex 2 Developer's Guide:*

- Chapter 50, "Understanding the Flex Data Management Service"
- Chapter 51, "Distributing Data in Flex Applications"
- Chapter 52, "Configuring the Data Management Service"

CHAPTER 20

Use ColdFusion Event Gateway Adapter

20

This tutorial shows you how to create a Flex application to send a message to a ColdFusion application and a ColdFusion component to send a message to a Flex application. The sample application does not take advantage of capabilities that are unique to Adobe Flex, instead, it describes the communication with ColdFusion applications that the ColdFusion Event Gateway Adapter enables.

To show the capabilities of the ColdFusion Event Gateway Adapter and the Flex Messaging event gateway, the sample application lets you enter information in a form in a Flex application. The Flex application sends the information through the ColdFusion Event Gateway Adapter and Flex Messaging event gateway to the ColdFusion application. The ColdFusion application then sends an e-mail message that contains the message to the recipient specified in the Flex application. Finally, the ColdFusion component sends a message to the Flex application, which displays the body of the message.

In this tutorial, you'll complete the following tasks:

Set up your development environment

The ColdFusion Event Gateway Adapter lets you create applications in which Flex Data Services and Macromedia® ColdFusion® MX 7.0.2 communicate. Flex Data Services includes the ColdFusion Event Gateway Adapter. ColdFusion MX 7.0.2 includes the Flex Messaging event gateway.

To complete this tutorial, you must have the following products installed:

- Flex Data Services
- ColdFusion MX 7.0.2

Start Flex and ColdFusion

To set up your development environment, you must start Flex Data Services and ColdFusion. This tutorial assumes that both Flex Data Services and ColdFusion are running on localhost (127.0.0.1) on your local computer. Because of the way in which the Remote Method Invocation (RMI) registry is created and maintained, Adobe recommends that you start Flex Data Services, and then start ColdFusion.

NOTE	The example ColdFusion application uses the cfmail tag. You must set up an e-mail server in the ColdFusion MX Administrator before testing the application.

Enable the ColdFusion Event Gateway Adapter

The messaging-config.xml file contains information about adapters and destinations, including network and server properties and channels. Generally, the file contains the following:

```
- service
  - adapters
    - adapter-definition
  - destination
    - properties
      - network
      - server
    - channels
```

TIP	To become familiar with the messaging-config.xml file, view it in an XML editor so that you can expand and collapse sections.

To ensure that Flex Data Services recognizes the ColdFusion Event Gateway Adapter, you edit the messaging-config.xml file, which is located in the C:\fds2\jrun4\servers\default\samples\WEB-INF\flex directory if you installed Flex Data Services using the default settings.

To enable communication through the Flex Messaging event gateway:

1. Copy the `<adapter-definition>` section in which `id="cfgateway"` appears from the sample messaging-config.xml file to the <adapter> section of the web application messaging-config.xml file.

   When you install Flex in the default location, the sample messaging-config.xml file is located in the C:\fds2\resources\config folder, and the web application messaging-config.xml file is located in the C:\fds2\jrun4\servers\default\samples\WEB-INF\flex folder.
2. Copy the `<destination>` section in which `id="ColdFusionGateway"` appears from the sample messaging-config.xml file to the web application messaging-config.xml file.
3. Save the file.

Create an instance of the Flex Messaging event gateway

To be able to communicate with the ColdFusion application through the Flex Event Gateway, you must create an instance of the gateway.

1. Create a blank file handleemail.cfc in the C:\CFusionMX7\wwwroot\flexgatewayexamples directory.

   (The flexgatewayexamples directory does not already exist.)
2. Start the ColdFusion MX Administrator.
3. Select Event Gateways > Gateway Instances.
4. Enter Flex2CF2 as the Gateway ID.
5. Select Flex Messaging - Flex as the Gateway Type.
6. Specify C:\CFusionMX7\wwwroot\flexgatewayexamples\handleemail.cfc as the CFC Path.
7. Select Automatic as the Startup Mode.
8. Click Add Gateway Instance.

Create the Flex application

The Flex application in this tutorial is a simple form in which you specify the elements of an e-mail message, including the recipient, the sender, the subject, and the message body.

Create a new MXML file

In this section, you create an MXML file in which the layout of user interface elements is exactly as you specify them, or absolute.

1. In an MXML editor, create a file that contains the following text:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*"
  layout="absolute"
   creationComplete="initApp()">

</mx:Application>
```

2. Save the file as flexemail2cf.mxml. in the C:\fds2\jrun4\servers\default\samples\dataservice\maypp folder.

Create the user interface

In this section, you create the controls to enter information to send an e-mail message:

1. Add the following MXML code after the <mx:Application> tag:

```
<mx:Consumer id="consumer" destination="ColdFusionGateway"
  message="messageHandler(event)"/>

<mx:TextInput x="103" y="13" width="291" id="emailto" editable="true"/>
<mx:TextInput x="103" y="43" width="291" id="emailfrom" editable="true"/
  >
<mx:TextInput x="103" y="73" width="291" id="emailsubject"
  editable="true"/>
<mx:TextArea x="103" y="102" width="291" height="236" id="emailmessage"
  editable="true"/>
<mx:Label x="63" y="15" text="To:" textAlign="right"/>
<mx:Label x="37" y="103" text="Message:" textAlign="right"/>
<mx:Label x="52" y="45" text="From:"/>
<mx:Label x="37" y="75" text="Subject:"/>
<mx:Button x="402" y="13" label="Send" id="emailsend"
  click="sendMessage();"/>
<mx:Label id="messagestatus" x="103" y="350" width="291" text="message
  not sent yet"/>
```

2. Save the file.

Import the required ActionScript classes

In this section, you create a script block and import a set of classes that you will use.

1. Create a script block for ActionScript code directly below the `<mx:Application>` tag:

```
<mx:Script>
  <![CDATA[

  ]]>
</mx:Script>
```

2. Directly below the `<![CDATA[` tag, add the following ActionScript import statements:

```
import mx.messaging.events.*;
import mx.messaging.Producer;
import mx.messaging.messages.AsyncMessage;
```

3. Save the file.

Create the Producer and Consumer

In this section, you declare the variables for the message Producer and the message Consumer.

1. Directly below the import statements in the script block, add the following variable definition:

```
public var pro:mx.messaging.Producer;
public var con:mx.messaging.Consumer;
```

2. Save the file.

Initialize the application

In this section, you create a function to create the message Producer.

1. Directly under the variable declaration, add the following method:

```
public function initApp():void {
  pro = new mx.messaging.Producer();
  pro.destination = "ColdFusionGateway";
  consumer.subscribe();
}
```

2. Save the file.

Send a message to ColdFusion

In this section, you create a function to send a message through the Flex Messaging event gateway to a ColdFusion application. You then create a structure, named msg, that contains the gateway ID, and the information necessary to send an e-mail message. The gateway ID is the ID you assign when you create the gateway instance in ColdFusion MX Administrator. Finally, you send the message to ColdFusion.

1. Directly below the initApp method, add the following code:

```
public function sendMessage():void {
  var msg:AsyncMessage = new AsyncMessage();

  msg.headers.gatewayid = "Flex2CF2";
  msg.body = new Object();
  msg.body.emailto = emailto.text;
  msg.body.emailfrom = emailfrom.text;
  msg.body.emailsubject = emailsubject.text;
  msg.body.emailmessage = emailmessage.text;

  pro.send(msg);
  messagestatus.text = "Message sent to ColdFusion.";
}
```

2. Save the file.

Receive a message from ColdFusion

In this section, you display a message sent from ColdFusion.

1. Add the following function after the sendMessage function:

```
private function messageHandler(event:MessageEvent):void {
  messagestatus.text = "Message received from ColdFusion";
}
```

2. Save the file as flexemail2cf.mxml. in the C:\fds2\jrun4\servers\default\samples\dataservice\maypp folder.

Verify that your code is correct

Your code should match the following code example. Verify that the content is correct.

```
<?xml version="1.0"?>

<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute"
   creationComplete="initApp()">

   <mx:Script>
```

```
  <![CDATA[
    import mx.messaging.events.*;
    import mx.messaging.Producer;
    import mx.messaging.messages.AsyncMessage;

    public var pro:mx.messaging.Producer;
    public var con:mx.messaging.Consumer;

    public function initApp():void {
      pro = new mx.messaging.Producer();
      pro.destination = "ColdFusionGateway";
      consumer.subscribe();
    }

    public function sendMessage():void {
      var msg:AsyncMessage = new AsyncMessage();

      msg.headers.gatewayid = "Flex2CF2";
      msg.body = new Object();
      msg.body.emailto = emailto.text;
      msg.body.emailfrom = emailfrom.text;
      msg.body.emailsubject = emailsubject.text;
      msg.body.emailmessage = emailmessage.text;

      pro.send(msg);
      messagestatus.text = "Message sent to ColdFusion.";
    }

    private function messageHandler(event:MessageEvent):void {
      messagestatus.text = "Message received from ColdFusion.";
    }
  ]]>
</mx:Script>

<mx:Consumer id="consumer" destination="ColdFusionGateway"
message="messageHandler(event)" />

<mx:TextInput x="103" y="13" width="291" id="emailto" editable="true"/
>
<mx:TextInput x="103" y="43" width="291" id="emailfrom"
  editable="true"/>
<mx:TextInput x="103" y="73" width="291" id="emailsubject"
  editable="true"/>
<mx:TextArea x="103" y="102" width="291" height="236"
  id="emailmessage" editable="true"/>
<mx:Label x="63" y="15" text="To:" textAlign="right"/>
<mx:Label x="37" y="103" text="Message:" textAlign="right"/>
<mx:Label x="52" y="45" text="From:"/>
<mx:Label x="37" y="75" text="Subject:"/>
<mx:Button x="402" y="13" label="Send" id="emailsend"
```

```
    click="sendMessage();"/>
  <mx:Label id="messagestatus" x="103" y="350" width="291" text="Message
    not sent yet."/>

</mx:Application>
```

Create the ColdFusion application

The ColdFusion application puts the information received from the Flex application in a structure. It then sends an e-mail message by using elements of the structure.

A ColdFusion application can handle data sent from a Flex application in either the header or the body of the message. The sample Flex application sends the data in the body of the message. To create the ColdFusion application, you create a ColdFusion component.

1. Create a blank file and enter the following code:

```
<cfcomponent
  displayname="Send e-mail from Flex application"
  hint="Handles incoming message from Flex">

  <!--- Handle incoming message. --->
  <cffunction name="onIncomingMessage" returntype="any">
      <cfargument name="event" type="struct" required="true">
    <!--- Create a structure to hold the message object from Flex. --->
    <cfset messagebody = event.data.body>

      <!--- Populate the structure. --->
      <cfset mailfrom="#messagebody.emailfrom#">
      <cfset mailto="#messagebody.emailto#">
      <cfset mailsubject="#messagebody.emailsubject#">
      <cfset mailmessage ="#messagebody.emailmessage#">

      <!--- Send the e-mail. --->
      <cfmail from="#mailfrom#"
        to="#mailto#"
        subject="#mailsubject#">
        <cfoutput>#mailmessage#</cfoutput>
      </cfmail>

      <!--- Create the structure to send back to Flex. --->
      <cfset success = StructNew()>
      <cfset success.body = "E-mail was sent at " & Now()>
      <cfset success.destination = "ColdFusionGateway">

      <!--- Send the structure to Flex. --->
      <cfset ret = SendGatewayMessage("Flex2CF2", success)>
  </cffunction>
</cfcomponent>
```

2. Save the file handleemail.cfc in the C:\CFusionMX7\wwwroot\flexgatewayexamples folder.

Test the application

To test the sample application, you must set up the testing environment, run the Flex application, and then view your e-mail client to ensure that the application sent the e-mail message successfully.

Set up the testing environment

Before testing the sample application, do the following:

- Ensure that Flex Data Services 2 is running.
- Ensure that ColdFusion is running.

TIP	To make debugging easier, you may want to start ColdFusion in a console by going to the CFusionMX7\bin directory and entering `cfstart`.

- Start the Flex2CF2 Flex Event Gateway instance.

To start the Flex2CF2 Flex Event Gateway instance:

1. Start the ColdFusion MX Administrator.
2. Select Event Gateways >Gateway Instances.
3. Click the Start button next to the Flex2CF2 gateway instance.

Run the application

To run the Flex application, you browse to the MXML file.

1. Open the http://localhost:8700/samples/dataservice/maypp/flexemail2cf.mxml file in a browser.
2. Enter a valid e-mail address in the To text box. Ensure that the e-mail address is one whose incoming e-mail you can check.
3. Enter the name of the sender in the From text box.
4. Enter the subject in the Subject text box.
5. Enter the message in the Message text area.
6. Click Send.

Check e-mail messages

To ensure that the application executed successfully, check the e-mail messages of the recipient specified in the Flex application. An e-mail message from the sender should be there, with the subject and body that you specified in the Flex application.

Index

E

F

G

H

I

J

L

M

P

S

W